INFORMATION AND COMMUNICATION TECHNOLOGIES
Dipartimento di Energia, Ingegneria dell'Informazione e Modelli Matematici (DEIM)
Settore Scientifico Disciplinare ING-INF/03

Mobile Edge Computing: Architetture ed Analisi della Live Migration

IL DOTTORE
Ing. Alessandro Randazzo

IL COORDINATORE
Prof. Alessandro Busacca

IL TUTOR
Prof.ssa Ilenia Tinnirello

CICLO XXXII
ANNO CONSEGUIMENTO TITOLO 2020

SOMMARIO

SINTESI 6
INTRODUZIONE 7
1 BACKGROUND 8
1.1 5G OVERVIEW 8
1.1.1 Wireless Sofware Defined Network 9
1.1.2 Network Function Virtualization 9
1.1.3 Onde Millimetriche 11
1.1.4 Massive MIMO 12
1.1.5 Multi RAT 13
1.2 Mobile Edge Computing (Mec) 13
1.2.1 Architettura ETSI Mobile Edge Computing 14
1.3 Virtual Machine e Containers 15
1.3.1 Docker Overview 17
1.4 Live Migration 18
1.4.1 Algoritmi di migrazione dati 19
1.5 Concetti di Memoria nei Sistemi Operativi Unix 21
1.5.1 File System Virtuale 21
1.5.2 Metriche Di Memoria 21
1.6 Distribuzione Statistica: Generalized Extreme Value (GEV) 23
1.7 Riepilogo 23
2 MOBILE EDGE COMPUTING (M-CORD) 24
2.1 Piattaforma M-Cord 24
2.1.1 Architettura Hardware M-Cord 25
2.1.2 Architettura Software CORD 26
2.1.3 XOS Framework 27
2.1.4 Servizio XOS 28
2.1.5 Composizione Di Un Servizio XOS 29
2.1.6 Mapping M-CORD su XOS 29
2.2 Riepilogo 31
3 ANALISI PRESTAZIONALI DI LIVE MIGRATION DI APPLICAZIONI MEMORY-INTENSIVE IN MEC 32
3.1 Introduzione 32
3.2 Architettura a Tre Livelli per la Service Migration 33
3.3 Architettura a Tre Livelli Modificata 34

3.4 Analisi dei Tempi di Migrazione per Ram-Intensive Application 36
3.5 Analisi dei Risultati Numerici 37
3.6 Riepilogo 39
4 RICONOSCIMENTO RISOLUZIONE VIDEO VIA METRICHE DI MEMORIA 41
4.1 Introduzione 41
4.2 Letteratura su Lavori Precedenti 43
4.3 Progettazione del Classificatore 43
4.3.1 Estrazione Dati e Stima del Working Set Size 43
4.3.2 Feature Analysis 44
4.3.3 Disegno di una Rete Neurale Multilayer Perceptron 46
4.4 Valutazione delle Prestazioni 48
4.4.1 Data Sets 48
4.4.2 Ambiente di Test e Software 48
4.4.3 Risultati Numerici 49
4.5 Riepilogo 52
5 OTTIMIZZAZIONE PRE-COPY: ARCHITETTURA RATAMCA 53
5.1 Introduzione 53
5.2 Letteratura su Lavori Precedenti 54
5.2.1 Architetture MEC 54
5.2.2 Stato dell'arte: Ottimizzazione PRE-COPY 54
5.3 Soluzione Proposta 55
5.3.1 Architettura di Rete 55
5.3.2 Modello di Simulazione 57
5.4 Valutazione delle Prestazioni 58
5.4.1 Modelli di Traffico 58
5.4.2 Scenari di Test 59
5.4.3 Risultati Numerici 60
5.5 Riepilogo 65
6 KATA CONTAINERS: UN' ARCHITETTURA EMERGENTE PER ABILITARE IN MODALITÀ VELOCE E SICURA I SERVIZI MEC 66
6.1 Introduzione 66
6.2 Kata Containers 67
6.2.1 Architettura 67
6.2.2 Networking 72
6.2.3 Storage 72
6.3 Analisi Qualitativa: Confronto Runc Vs Kata-Runtime 73

6.4 Riepilogo 75
7 CONCLUSIONI 76
PUBBLICAZIONI 78
APPENDICE 79
APPENDICE A 79
APPENDICE B 81
APPENDICE B1 81
APPENDICE B2 85
APPENDICE C 86
BIBLIOGRAFIA 87

RINGRAZIAMENTI

Con queste poche righe vorrei ringraziare tutti coloro che chi mi hanno sostenuto nel mio percorso di ricerca. In primo luogo quindi è necessario menzionare il tutor di questo lavoro di tesi, la professoressa Ilenia Tinnirello, che mi ha fornito indicazioni e correzioni preziose senza le quali questo lavoro non avrebbe potuto vedere la luce, ma anche quella libertà, che non tutti permettono, di seguire i propri interessi di ricerca. Ciò mi ha permesso di vivere questo percorso accademico da studente curioso ed entusiasta. Grazie di cuore Ilenia! Desidero inoltre ringraziare tutto il personale dell'università di Palermo con il quale ho dovuto interagire durante questi tre anni.

Un sentito grazie ai miei familiari che hanno come sempre appoggiato la mia scelta e sostenuto. In particolare dedico questo traguardo a mio papà che in quest'ultimo periodo non è stato benissimo e spero di avergli regalato un altro momento di gioia ed a lui che la dedico.

"Tra vent'anni sarai più infastidito dalle cose che non hai fatto che da quelle che hai fatto. Perciò molla gli ormeggi, esci dal porto sicuro e lascia che il vento gonfi le tue vele. Esplora. Sogna. Scopri."

Mark Twain

SINTESI

Con l'ormai prossima rete mobile 5G entreranno a far parte della nostra quotidianità nuovi servizi applicativi, mai prima possibili, grazie all'avvicinamento di risorse di calcolo e di memoria nei pressi dell'utente in mobilità. Un'architettura abilitante i futuri servizi è quella di Mobile Edge Computing (MEC) in cui cloud di capacità inferiori rispetto a quelli presenti nella core della rete sono dislocati nei pressi della stazioni radio e metteranno a disposizione risorse di calcolo tali da permettere, tramite la tecnica di offloading, la fruizione di servizi quali realtà aumentata, gaming online, contenuti streaming ad alta risoluzione ed operazioni di data analytics. Ogni nuovo paradigma porta con sé nuove sfide d'affrontare e risolvere per potere essere applicabile. Ecco, dunque, che s'introduce l'obiettivo perseguito durante questo percorso di ricerca che è stato quello di analizzare, modellare diversi aspetti di uno dei tanti problemi del MEC che è la continuità del servizio che è in funzione non solo del normale spostamento dell'utente mobile, ma anche dalle risorse presenti del server sorgente. Il percorso di studi e di ricerca è stato così strutturato: Studio iniziale sulle nuove sfide del 5G per poi dedicarsi ad uno studio approfondito sull'architettura di riferimento del mobile edge computing sino ad una piattaforma open-source di nome M-CORD. Questa prima parte è stata propedeutica per potere svolgere i seguenti lavori di ricerca. Un primo lavoro è consistito nel proporre una modifica di un framework per la live migration ed analizzare i tempi di migrazione di applicazioni RAM-intensive sviluppando un modello simulante l'esecuzione di tali applicazioni con uno dei possibili algoritmi di migrazione e cioè quello di pre-copy anzichè quello di post-copy. Le nostre ipotesi confortate dai risultati numerici hanno evidenziato che l'algoritmo di pre-copy è preferibile nella service migration di applicazioni RAM-intensive. Un successivo lavoro è nato dalla considerazione che grazie ai server MEC vi sarà un'altissima diffusione di contenuti multimediali e molti di essi saranno a pagamento e regolati da contratti di service level agreement (SLA) e, dunque, abbiamo progettato e sviluppato un'applicazione di machine learning che potrebbe essere installata direttamente nello smartphone che in background analizza ed individua in tempo reale la risoluzione video ricevuta così da verificare il rispetto del SLA. Inoltre abbiamo anche disegnato un'architettura in cui la service migration possa essere originata dal client mobile, una volta verificatosi la violazione (breach) contrattuale. I risultati sono stati promettenti visto che si è ottenuta un'accuratezza superiore al 95% su dati mai visti. Questo lavoro ha anche evidenziato le enormi potenzialità offerte dal machine learning capace di discernere caratteristiche intrinseche dell'applicazione senza alcun ausilio di analisi protocollare. Un terzo lavoro è stato quello di estendere il modello di analisi dei tempi di migrazione per applicazioni RAM-intensive andando a simulare, sempre sotto l'ipotesi di applicare l'algoritmo di pre-copy, la presenza di un controller SDN che permette una continua riconfigurazione dinamica dei percorsi di rete. A tale scopo abbiamo quindi proposto una nuova architettura basata sia su MEC che SDN, modellato e simulato alcuni scenari di traffico di rete ed analizzato le prestazioni temporali. I risultati hanno evidenziato che grazie alla continua riconfigurazione dinamica dei percorsi, assicurando quello più veloce ad ogni iterazione dell'algoritmo, in determinati scenari si sono ottenuti tempi di migrazioni più bassi sino al 90%. È, dunque, chiaro che la cooperazione di MEC e SDN possa fornire un servizio di migrazione più efficiente poiché ad un minore tempo di trasferimento permetterà il rispetto dei stringenti vincoli temporali di molte applicazioni. Infine, un quarto lavoro è nato dall'osservazione che se è vero che la tecnologia più in linea per abilitare i nuovi servizi 5G sia quella dei containers, essendo molto più veloci delle virtual machine, è altrettanto vero che ad oggi per design i containers sono soggetti ad intrinsechi problemi di sicurezza dovuto principalmente alla condivisione del Kernel host con tutti i containers in esecuzione. Per tale motivo, anziché sviluppare soluzioni personalizzate non standardizzate che facessero uso sia dei containers che delle virtual machine, abbiamo ricercato ed analizzato una nuova promettente architettura che potrebbe costituire la base per la migrazione dei servizi garantendo al contempo sia leggerezza e velocità che sicurezza di nome Kata-containers. Essendo una nuovissima architettura abbiamo svolto uno studio sulle sue principali caratteristiche e li abbiamo poi confrontate da un punto di vista qualitativo con quelle di Docker che rappresenta lo standard de facto per la tecnologia containers. Da tale analisi è emerso che tale nuova architettura, se pur ancora da migliorare, potrà rappresentare il pilastro portante per la migrazione dei servizi tra MEC server.

INTRODUZIONE

Lo studio e ricerca che si è voluta intraprendere durante il triennio è stato quello di focalizzarsi su alcuni aspetti della migrazione del servizio nelle future reti 5G. La motivazione principale nasce dal fatto che la mole di dati multimediali in mobilità continuerà a crescere con un trend esponenziale favorito anche dall'introduzione di nuovi casi d'uso come per esempio realtà aumentata, giochi online e contenuti in alta definizione. Si stima che il consumo di dati in mobilità continuerà ad aumentare, nonostante l'uso di meccanismi di offload. Ericsson ritiene che l'utilizzo medio mensile di dati in mobilità per utente nel Nord America oggi sia intorno a $8{,}6\ GBytes$ e raggiungerà i $50\ Gbytes$ entro la fine del 2024 (1). Molto potrebbe dipendere dal fatto che ci sarà un uso significativo di applicazioni di realtà aumentata. Basti pensare che soltanto 10 minuti di utilizzo di un'applicazione di realtà aumentata ogni giorno potrebbero tradursi in 50 GB al mese per utente. Ecco, quindi, che una migrazione dei dati che sia più fluida possibile durante la fruizione di un servizio diventerà fondamentale per tutti gli operatori di servizi che intenderanno fornire contenuti di alta qualità. Partendo, dunque, da tale presupposto, abbiamo analizzato e modellizzato alcune problematiche relative alla migrazione di particolari applicazioni, definite RAM-Intensive. Abbiamo inoltre, proposto modifiche ad un framework per la migrazione dei servizi al fine di ottimizzare i tempi di migrazione tra sorgente e destinazione. Abbiamo pure immaginato uno scenario in cui un utente possa controllare istante per istante, mediante tecniche di machine learning, la qualità video pattuita col fornitore di servizi e far scattare una richiesta di cambio sorgente verso la rete nel momento in cui le condizioni non dovessero più rispettare i requisiti pattuiti. Ancora, abbiamo proposto un'architettura in cui la cooperazione di Mobile Edge Computing e Software Defined Network possa ridurre i tempi di migrazioni sfruttando la riconfigurazione dinamica dei percorsi. Infine, abbiamo evidenziato come la tecnologia da considerare per potere operare migrazioni in linea con gli stringenti requisiti di molte applicazioni siano i containers ed in particolare abbiamo introdotto e descritto una nuovissima architettura il cui nome è Kata Containers che risulta essere un ibrido tra tra virtual machine (VM) e containers ovvero cerca di prendere la robustezza e la sicurezza delle VM e la leggerezza e velocità tipica dei containers. Riassumendo, la tesi è stata così strutturata: Primo capitolo Background su vari argomenti propedeutici per potere seguire i lavori svolti, secondo capitolo studio di una piattaforma di Mobile Edge Computing, terzo introdotto un modello di analisi dei tempi e definizione di un framework per la migrazione di applicazioni RAM-Intensive, quarto analisi, progettazione e sviluppo di un classificatore, basato su reti neurali per la risoluzione video in tempo reale, quinto estensione del modello di analisi dei tempi per migrazioni di applicazioni RAM-Intensive con l'ausilio di un modello di controller SDN, sesto introduzione ed analisi qualitativa dell'architettura Kata Containers come possibile soluzione standard nella migrazione del servizio in mobilità. Infine ultimo capitolo le conclusioni.

1 BACKGROUND

Questo primo capitolo introduce tutta una serie di concetti indispensabili per seguire i lavori svolti durante il triennio di studio e ricerca.

1.1 5G OVERVIEW

La rete 5G presenterà nuove sfide che richiederanno non solo maggiori data rates, ma anche ultra low latency, high reliability e security. Per potere raggiungere tali obiettivi è stato necessario definire un nuovo standard che include lo sfruttamento di bande di frequenza nel campo delle onde millimetriche e di una nuova architettura di rete. La rete 5G non sarà soltanto un miglioramento della velocità nel trasferimento dati, ma anche di altri requisiti, come per esempio, 'ultra low latency' che permetterà di fornire una interattività in tempo reale con i servizi attraverso l'utilizzo di sistemi cloud. La tecnologia 5G può essere sintetizzata da 7 specifici requisiti:

- Fino a 10Gbps data rate
- 1000x larghezza di banda per unità d'area
- Fino a 100x numero di dispositivi connessi per unità d'area (rispetto al 4G LTE)
- 99.999% availability
- 100% coverage
- 90% riduzione del consumo di energia nelle reti
- Fino a 10 anni di durata della batteria per dispositivi IoT low power

Come si può capire poter soddisfare tutti questi requisiti coinvolge tutti i livelli protocollari a partire dal livello fisico sino a quello applicativo. Bisogna sottolineare che a secondo del caso d'uso che si dovrà sviluppare solo un sotto insieme di questi sette requisiti saranno necessari. Questi requisiti sono giustificati dal fatto che secondo un report Cisco (2) si prevede un notevole incremento sia di dispositivi connessi che traffico dati mobile. Come mostrato nella Figura 1

Figura 1 -Trand Dispositivi Mobili e Traffico Dati

Per potere raggiungere gli obiettivi che la nuova rete 5G si propone di soddisfare sarà necessario introdurre nuove tecnologie quali per esempio:

- Wireless Software Defined Network
- Network Function Virtualization
- Onde Millimetriche
- Massive MIMO
- Multi-RAT

1.1.1 WIRELESS SOFWARE DEFINED NETWORK

La Wireless Software Defined Network (WSDN) è un'evoluzione del paradigma SDN (Software Defined Network), il cui principio è la realizzazione via software di funzioni di rete di basso livello con l'intento di avere una rete più flessibile ottenuta mediante astrazione software.

L'idea si basa nel separare il piano di controllo (Control Plane) dal piano dati (Data Plane) separando il sistema in due parti: quello che prende le decisioni relative all'invio del traffico, da quello che si occupa esclusivamente dell'inoltro dei pacchetti a destinazione. Questa separazione rende l'architettura di rete altamente adattabile e dinamica. Nel caso delle reti 5G la WSDN non è altro che SDN portata in ambiente mobile con la finalità di poter soddisfare i diversi casi d'uso con differenti requisiti di Quality of Service (QoS). La rete 5G non sarà più, dunque, basata sulle classiche tecnologie di routing e switching (3). Inoltre, sicurezza, resilienza, robustezza e integrità dei dati saranno una tra le prime priorità nella progettazione delle future reti software (4) . L'infrastruttura di rete wireless sarà basata su SDN, che fornirà la comunicazione tra servizi, applicazioni cloud e il dispositivo mobile. Dunque, la rete potrà essere gestita sulla base di necessità real-time e verrà dinamicamente riconfigurata sfruttando le risorse virtualizzate. Un esempio di generica rete 5G basata su SDN è mostrata in Figura 2. (5)

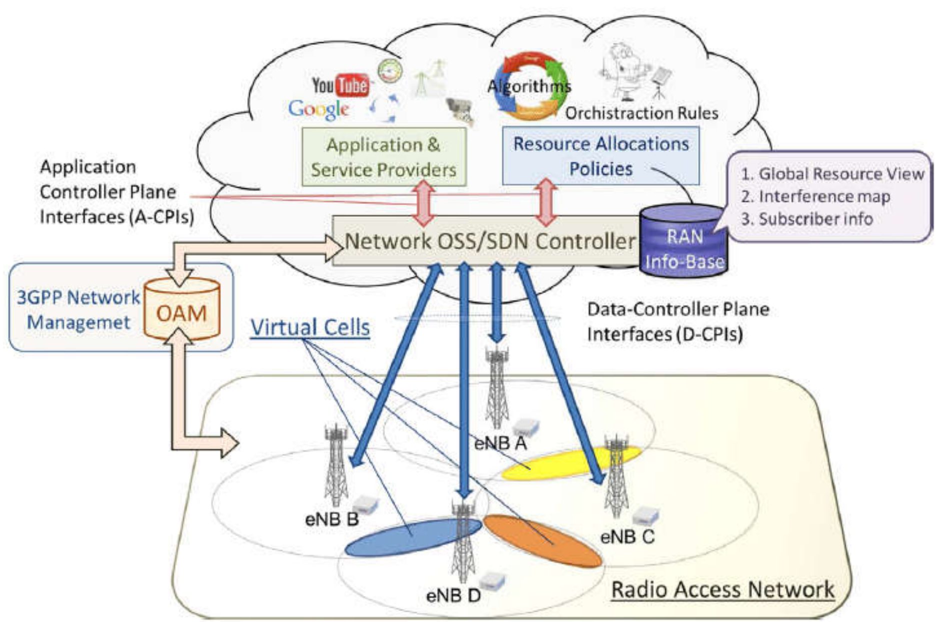

Figura 2 - Rete 5G basata su SDN

1.1.2 NETWORK FUNCTION VIRTUALIZATION

Altra tecnologia abilitante le reti 5G è la Network Function Virtualization (NFV) che è complementare a SDN e permette la virtualizzazione di specifiche funzionalità di rete che prima erano strettamente legate all'hardware in cui giravano mentre grazie alla virtualizzazione si astraggono da esso e sono eseguite su un'infrastruttura cloud. Disaccoppiando le funzioni di rete dall'hardware la rete 5G acquisisce quell'agilità, scalabilità e flessibilità che mai prima di ora era stata raggiunta da una rete mobile. A tali vantaggi si aggiungeranno riduzioni di spese capitali (CAPEX) visto che non sarà più necessario utilizzare determinato hardware proprietario per avere una specifica funzionalità di rete, ma anche di costi operativi e gestionali (OPEX) grazie

all’aggregazione delle risorse per le funzioni di rete virtuale che sono eseguite presso sistemi cloud opportunamente orchestrati. Attraverso NFV la core network diventa più intelligente, scalabile, elastica e flessibile attraverso la virtualizzazione di varie entità come per esempio MME, PGW e SGW abbattendo le distanze geografiche tra tali elementi di rete, poiché una volta virtualizzati, per esempio SGW e PGW, possono essere co-locati con la base station. Si può, dunque, affermare che grazie a NFV la core network diventa programmabile e i servizi potranno essere altamente personalizzati a seconda del tipo di richiesta mediante la creazione on-the-fly di network slice. Per maggiori approfondimenti si rimanda il lettore al capitolo successivo. La Figura 3 mostra un esempio di slicing in cui sopra l’infrastruttura fisica sono realizzate differenti reti virtuali (slices) per l’esecuzione di differenti casi d’uso aventi differenti requisiti in termini di banda, data rate, latenza, affidabilità e sicurezza.

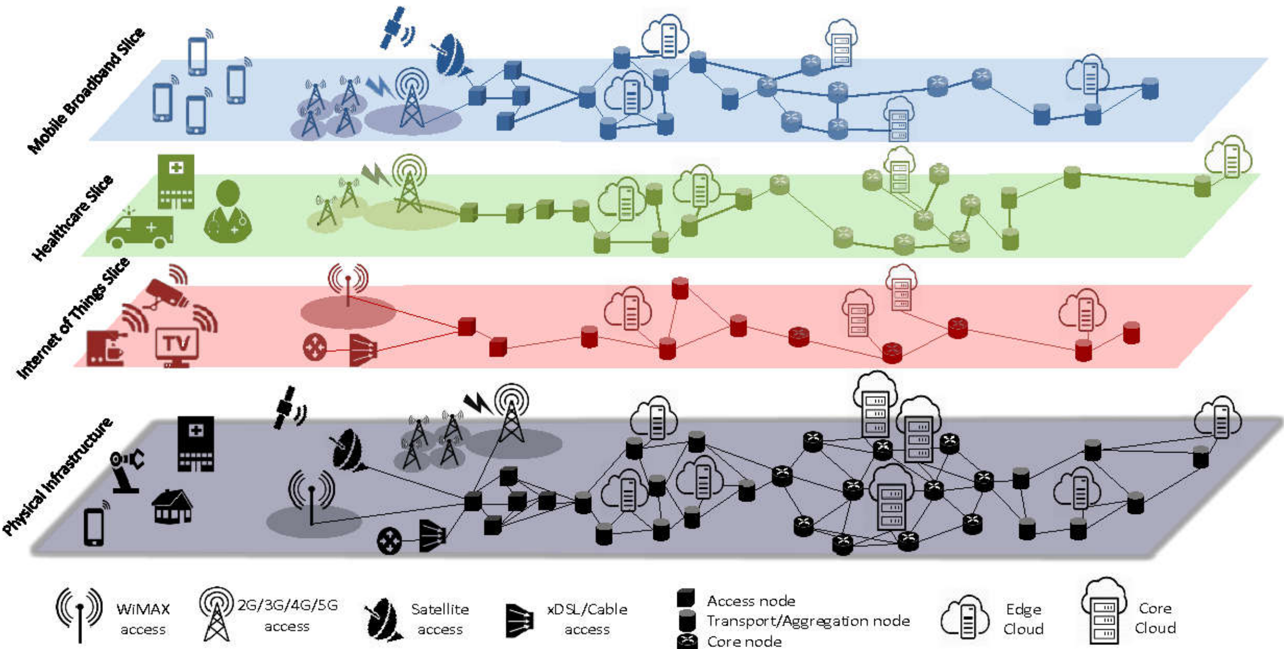

Figura 3 - 5G Network Slicing

1.1.3 ONDE MILLIMETRICHE

La richiesta di elevati data rate e bassissime latenze si scontra con la scarsità di risorse spettrali e quindi larghezza di banda disponibile. Per tali motivi sono iniziate ricerche nella banda di frequenza tra 30 – 300 GHz che sono quelle millimetriche. Esse rappresentano, ad oggi, una grande disponibilità di risorse spettrali sia senza licenza che con. Una delle sfide per poter sfruttare tali frequenze è l'assorbimento di energia causato dall'atmosfera, dalla pioggia e dalla neve causando una forte limitazione nella distanza di propagazione del segnale. L'effetto di blocco delle frequenze ad onde millimetriche è inevitabile, quindi la ricerca punta a sfruttarle in ambienti indoor e comunque con base station con raggio di copertura di alcune centinaia di metri. Una soluzione che compensa questa elevata attenuazione è quella di usare array di antenne attuando quello che prende il nome di multiplexing spaziale in cooperazione con il beamforming che con l'uso di opportuni pesi per ogni stream di dati riesce per ogni elemento d'antenna a pilotare la direzione verso cui gli stream sono trasmessi in modo da limitare al minimo l'interferenza verso dispositivi vicini, ma estranei alla corrente trasmissione. Grazie allo sfruttamento delle onde millimetriche nasceranno nuovi modelli di comunicazione come per esempio la comunicazione diretta tra i dispositivi o scenari di comunicazione di guida autonoma. La Figura 4 (6) mostra un esempio di futura comunicazione 5G.

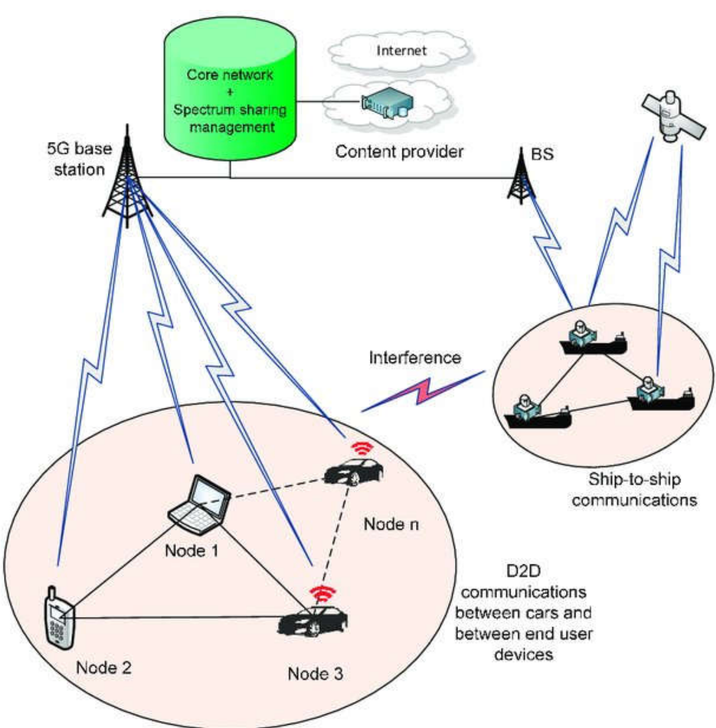

Figura 4 - - Comunicazione dispositivo a dispositivo (D2D) in 5G

1.1.4 MASSIVE MIMO

Altra tecnologia abilitante nel 5G sarà il Massive MIMO evoluzione dell'attuale multi-user MIMO (MU-MIMO) sviluppato per i sistemi 4G, in cui solo alcune decine di antenne sono integrate sulle base-station e nei dispositivi mobili utente. Uno degli obiettivi del Massive MIMO è quello di aumentare sia la capacità che il throughput di sistema. Il numero di antenne previste sulla base-station sarà dell'ordine delle centinaia e saranno previsti diversi layout di antenna da quello ad array rettangolare a quello lineare sino a quello cilindrico al fine di coprire diversi scenari di comunicazione. Generalmente gli array lineari e rettangoli andranno collocati in cima a palazzi molto alti, come grattacieli, di modo che gli utenti che abitano nei piani più alti possano godere di ottima copertura indoor, mentre quelle cilindriche sono progettate per utenti in mobilità. I principali vantaggi dell'utilizzo massivo di antenne saranno una migliore efficienza spettrale ovvero a parità di banda disponibile una maggior quantità di dati inviati al secondo, ma anche latenze ridotte grazie all'elevato numero di flussi trasmessi contemporaneamente. Dato l'elevato numero di antenne per la stima del canale in downlink verrà sfruttato il principio di reciprocità dell'uplink e del downlink . Infatti sarebbe troppo oneroso per un terminale mobile stimare il canale caratterizzato da centinaia di segnali provenienti dalla base-station. Quindi il terminale invierà alcuni pilot per la stima del canale in uplink e la base station sulla base di essi stimerà quello in downlink. Infine grazie al Full-Dimension MIMO (FD-MIMO) (7) che utilizza un modello di canale 3D saranno possibili performance migliori. Infatti aumentando il grado di libertà portato dai flussi di antenne in altezza, la capacità del sistema potrà essere migliorata. La Figura 5 mostra un esempio di Massive MIMO.

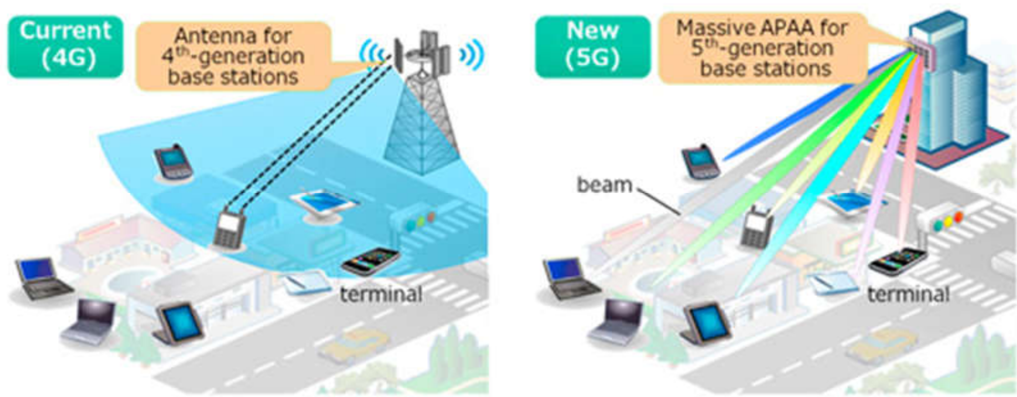

Figura 5 - Massive MIMO

1.1.5 MULTI RAT

Ultima tecnologia abilitante il 5G che vogliamo descrivere è il Multi-RAT (8) cioè l'integrazione di più tecnologie di accesso radio, di diverse bande di frequenza, oltre alle onde millimetriche già introdotte, che possono fornire una larghezza di banda considerevole come quelle al di sotto dei 6 GHz. Infatti è ben noto che un modo per aumentare il throughput utente e la capacità del sistema sarebbe quella di avere a disposizione di più ampie larghezze di banda. Dalle ricerche emerge che le frequenze di poco al di sotto dei 6 GHz supportano una copertura più vasta rispetto le onde millimetriche e collegamenti wireless più stabili così che tali frequenze potrebbero essere utilizzate per messaggi di segnalazione come paging. La vera sfida sarà l'integrazione e l'interoperabilità di più RAT con diverse bande di frequenza nella futura rete 5G. La Figura 6 mostra una possibile soluzione di integrazione tra onde millimetriche e frequenze intorno ai 6 GHz.

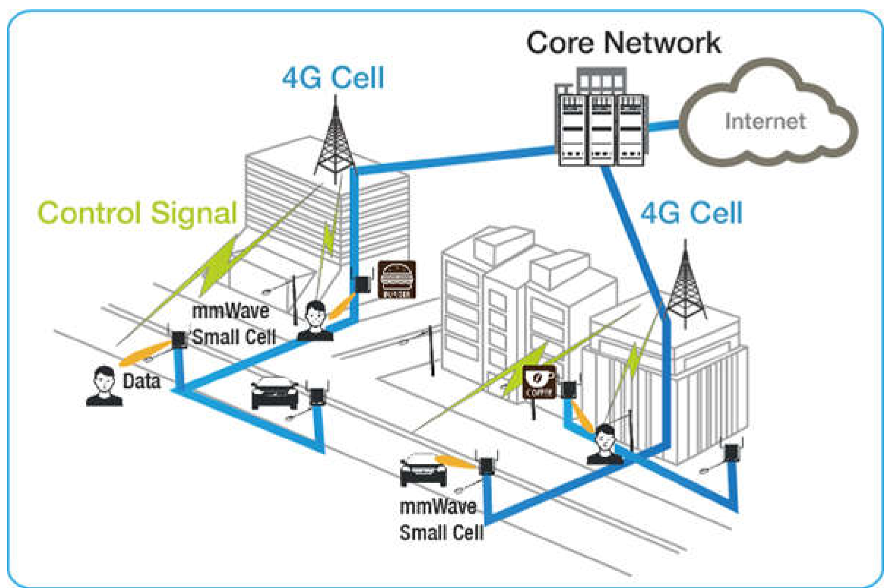

Figura 6 - Multi-RAT 5G

1.2 MOBILE EDGE COMPUTING (MEC)

Un'altra importante tecnologia che permetterà la realizzazione di importanti casi d'uso, come la comunicazione di milioni di dispositivi IoT (Internet of Things), quali sistemi smart di riscaldamento, veicolari e perfino presenti nei nostri vestiti è il Mobile Edge Computing. I principali obiettivi di tale tecnologia sono: *'Ottimizzazione delle risorse mediante tecnica di computation offloading, Ottimizzazione del trasferimento dati dall'edge al core della rete mediante Big Data analityc eseguita nell'adge, Abilitazione di nuovi servizi portando le risorse sia hardware che fisiche in prossimità degli utenti mobili, servizi di tipo context-aware tramite le informazioni provenienti dalla RAN'* Nello scenario di milioni di dispositivi IoT tra loro comunicanti, se pur vero che da soli, producono una contenuta, ma continua mole di dati, la loro presenza contemporanea nella rete produrrà una grandissima quantità totale di dati che bisognerà analizzare entro stringenti intervalli di tempi oppure richiederanno un ingente quantità di banda a secondo del caso d'uso. Per poter raggiungere tale obiettivo sarà necessario avere ingenti risorse di calcolo in termini di processori, disco, memoria principale e software il più vicino possibile alle sorgenti e cioè sia ai dispositivi IoT che ai classici terminali mobili quali smartphone e tablet. Ecco, quindi,

che tecnologie già ben consolidate in ambito cloud data-center, come Amazon Elastic Compute e Microsoft Azure devono in qualche modo essere traslate in prossimità del host passando da un'infrastruttura centralizzata a quella distribuita. Si può affermare che il MEC rappresenta la perfetta fusione del mondo TLC relativamente all' accesso di rete mobile con quello IT basato su infrastrutture cloud e non più come fatto sino ad oggi due tecnologie distinte e separate. A differenza dei server cloud centralizzati i MEC server sono gestiti localmente dall'operatore di rete. Inoltre le risorse di calcolo sono virtualizzate all'interno dei mobile edge hosts ed esposte attraverso application program interfaces (APIs), così che esse possano essere accessibili sia all'utente che alle stesse applicazioni. Si può, dunque, affermare che con l'avvento del MEC la rete mobile evolverà da semplice funzione di comunicazione puramente passiva a diventare parte integrante di un nuovo paradigma di comunicazione basato sul calcolo e fornitura delle risorse di calcolo. Maggiori dettagli riguardanti il MEC saranno oggetto del successivo capitolo. La Figura 7 mostra in maniera semplificata il concetto di MEC.

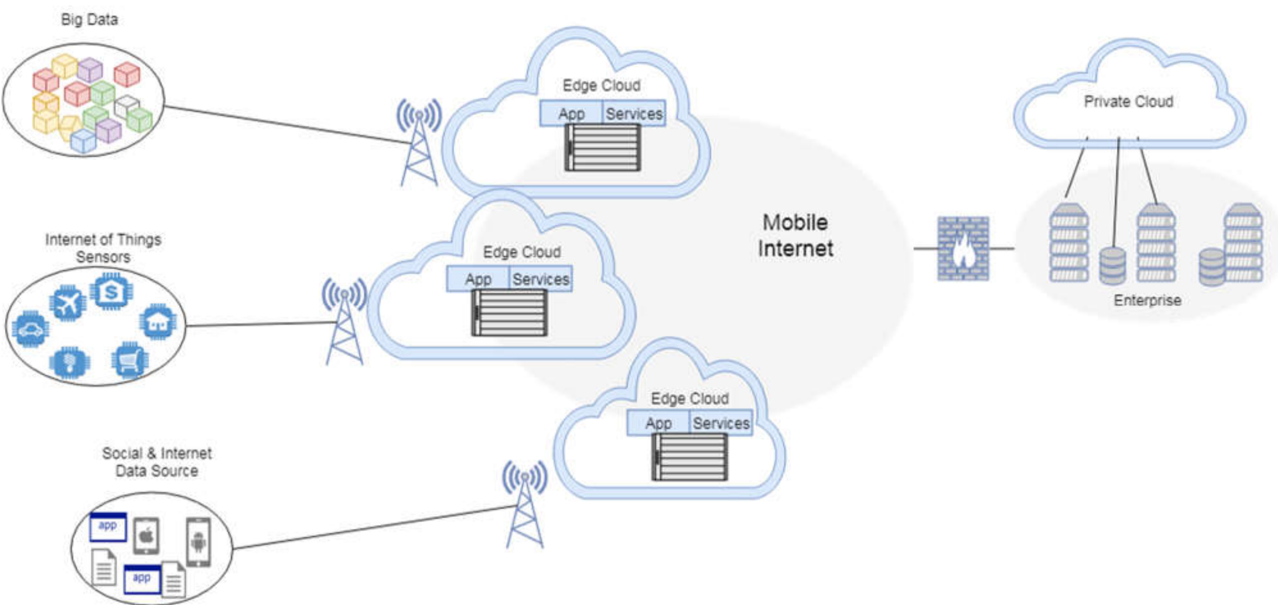

Figura 7 - Generico Scenario MEC

1.2.1 ARCHITETTURA ETSI MOBILE EDGE COMPUTING

Il termine 'Mobile Edge Computing' fu standardizzato dalla European Telecommunications Standards Institute (ETSI) e dalla Industry Specification Group (ISG). Secondo ETSI mobile edge computing è così definito : *'Mobile Edge Computing provides an IT service environment and cloud computing capabilities at the edge of the mobile network, within the Radio Access Network (RAN) and in close proximity to mobile subscribers.'* Gli obiettivi che si pone di affrontare sono:

1. Ottimizzare la risorsa mobile tramite hosting di applicazioni computing-intensive
2. Abilitare servizi cloud in prossimità dell'utente mobile
3. Fornire servizi context-aware con l'aiuto delle informazioni della RAN
4. Ottimizzare la grande mole dei dati prima di inviarli al cloud

L'architettura MEC di riferimento è mostrata in Figura 8. Le applicazioni sono eseguite come virtual machine o containers sopra l'infrastruttura di virtualizzazione ed interagiscono con la

'Mobile Edge Platform' per eseguire procedure di supporto relativamente al ciclo di vita dell'applicazione.

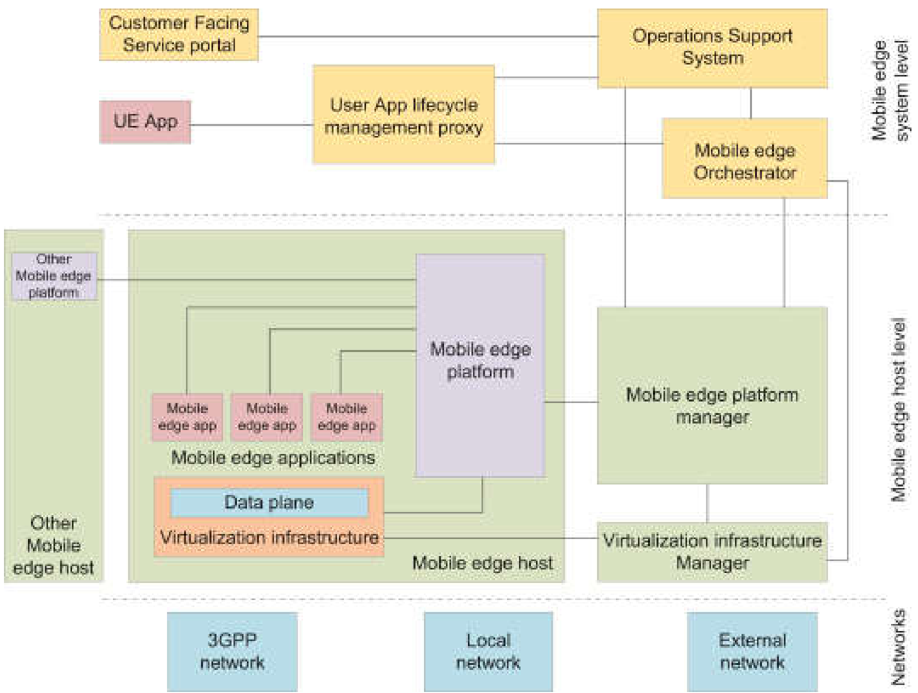

Figura 8 - MEC Architecture (Fonte: ETSI)

La '*virtualization infrastructure*' include anche un 'data plane' che esegue le regole di traffico che gli sono state impartite dalla *'mobile edge platform*' la quale le instrada tra le applicazioni o a reti locali o esterne. La parte di '*Mobile Edge Host Level*' comprende la '*Mobile Edge Platform Manager*' e la '*Virtualization Infrastructure Manager*'. La prima gestisce il ciclo di vita delle applicazioni come i requisiti di autorizzazione, regole di traffico, configurazioni DNS. La seconda è responsabile di allocare risorse, gestirle e rilasciarle nell'infrastruttura virtualizzata. Nella parte superiore e cioè nel '*Mobile Edge System Level*' l''*Operation Support System*' riceve richieste dalle applicazioni utente tramite un *'Management Proxy'* oppure tramite un portale da utenti di operatori di terze parti. Tale sistema di supporto è preposto a decidere se accettare o meno la richiesta. Quelle accettate passano al '*Mobile Edge Orchestrator*' per ulteriore processing. Si può affermare che l'orchestratore MEC è il cuore pulsante di tutta l'architettura poiché mantiene una visione generale e globale sui MEC host, risorse e servizi MEC disponibili e topologia.

1.3 Virtual Machine e Containers

Le tecnologie abilitanti la migrazione dei servizi sono principalmente due: Virtual Machine e Containers. Sebbene entrambe le soluzioni permettono l'esecuzione di multiple ed isolate applicazioni sopra un dato sistema operativo, detto host, esse presentano delle differenze che nel caso della live migration possono essere determinanti al fine di ottenere una migrazione del tutto trasparente al client mobile. Le virtual machine emulano il comportamento di un'intera macchina fisica dall'hardware sino al software applicativo attraverso l'astrazione di tutte le risorse fisiche disponibili della macchina host. Ciò è reso possibile tramite una virtual machine manager, chiamato '*hypervisor'*, che agisce come orchestratore delle varie virtual machine create garantendone l'isolamento e la loro gestione. Tipicamente le virtual machine sono eseguite a livello applicativo e, dunque, non è possibile l'esecuzione di istruzioni privilegiate sulle CPU fisiche, perfino quando i programmi virtuali richiedono l'esecuzione di istruzioni in kernel-mode. Una tipica soluzione si

basa sui *'trap'*, ossia generare degli errori fittizi così da costringere hypervisor ad eseguire queste istruzioni privilegiate. (9). Tali soluzioni dato che consumano diversi cicli CPU non sono efficienti. Negli ultimi anni sono state introdotte nuove soluzioni a livello hardware , come Intel-VT (10) e AMD-V (11) allo scopo di ridurre questo overhead.

L'hypervisor a seconda di dove è situato tra l'hardware del sistema host e le virtual machine è definito come:

1. **'Tipo 0'**: Di tipo hardware
2. **'Tipo 1'**: Hypervisor implementato come sistema operativo
3. **'Tipo 2'**: Hypervisor eseguito come un'applicazione sopra il sistema operativo host

Quello di *'Tipo 0'* sono soluzioni hardware proprietarie di solito presenti nei mainframe. Essi garantiscono ottime prestazioni, ma al contempo sono troppo rigide per deployment su larga scala. '*Tipo 1*' è uno strato software direttamente sopra l'hardware della macchina host e funge da sistema operativo, così d'avere totale accesso fisico alle CPU. In altre parole opera direttamente in kernel mode. Un ben noto hypervisor di *'Tipo 1'* è KVM (12) implementato come un modulo kernel. Infine *'Tipo 2'* sono delle normali applicazioni in esecuzione (processi) sopra il sistema operativo host. Un noto hypervisor di 'Tipo 2' è Virtual Box (13). In generale le prestazioni di quest'ultimo tipo di hypervisor sono più scadenti rispetto ai primi due visto che esso funziona come una normale applicazione in user-mode.

Un'altra tecnologia di virtualizzazione che negli ultimi si è diffusa sempre di più è quella dei Containers. Essa viene anche etichettata come 'Lightweight Virtualization' poiché non astrae l'intero hardware per emulare un pc completo dall' hardware al sistema operativo, kernel, librerie, binari ed applicazioni, ma piuttosto virtualizzano soltanto il sistema operativo. In generale ogni container condivide solo l'utilizzo del kernel del sistema operativo host e, di solito, anche librerie e binari. I componenti del sistema operativo vengono condivisi in sola lettura e la condivisione di componenti software come le librerie riduce in maniera drastica la necessità di duplicare il codice del sistema operativo mantenendo in esecuzione più istanze in contemporanea. Ecco spiegato il motivo per cui i containers sono quindi eccezionalmente leggeri, pesano pochi megabyte e impiegano pochi secondi per avviarsi al contrario, evidentemente, delle macchine virtuali che hanno un intero stack protocollare replicato oltre ad un loro proprio sistema operativo. Inoltre in termini di risorse utilizzate, i containers consumano esattamente le medesime risorse consumate dal sistema operativo host e delle applicazioni installate. In buona sostanza i containers non sono altro che dei processi guest direttamente in esecuzione sul kernel host condiviso in modalità user-mode senza, dunque, la necessità di un hypervisor e di un proprio sistema operativo come invece fanno le virtual machine. In realtà anche un container può aver un proprio sistema operativo, ma molto più leggero di quello di una macchina virtuale, poiché ne condividerà sempre il kernel del sistema operativo host. Questa tipologia di container prende il nome di *'system container'* che hanno lo scopo di mandare in esecuzione multipli servizi contemporaneamente a differenza degli *'application container'* come per esempio Docker (14) che sono in rapporto uno ad uno container e applicazione. La Figura 9 riepiloga le differenze tra queste due differenti tecnologie di virtualizzazione.

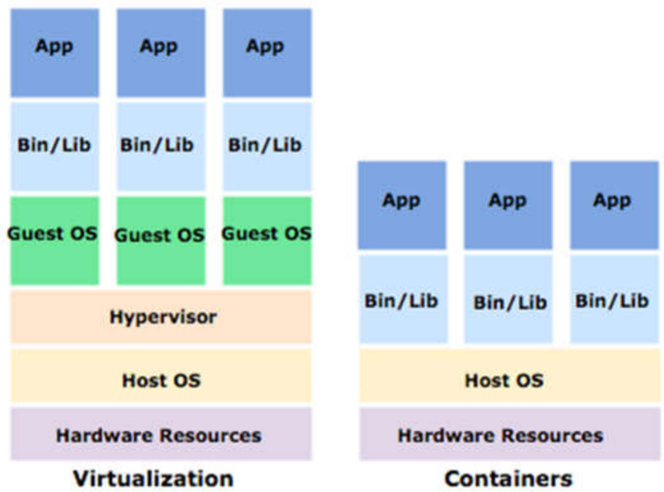

Figura 9 - Confronto VM - Container

1.3.1 DOCKER OVERVIEW

Docker è una piattaforma software open source per creare, installare e gestire applicazioni che girano dentro i containers su un comune sistema operativo. La prima versione di Docker, rilasciata nel 2013, era un blocco monolitico che inglobava operazioni di runtime sia ad alto che a basso livello. Nel 2015, sotto la pressione di 'Open Container Initiative' (OCI) (15), un progetto open container il cui scopo è creare uno standard industriale aperto riguardo il formato delle immagini dei containers e del loro runtime, Docker introdusse tale formato a partire dalla versione 1.11. Questa release comportò un 'refactoring' massivo dell'engine per poterlo renderlo OCI compliant. I principali componenti di Docker a partire dalla versione 1.11 sono quattro:

1. Dockerd: Un demone in ascolto a richieste API provenienti dagli utenti tramite riga di comando (CLI)
2. Containerd: Un demone che controlla 'runc' e gestisce l'intero ciclo di vita del container (immagini, storage, esecuzione tramite runc e rete)
3. Containerd-shim: Un demone che permette l'esecuzione di containers 'deamon-less'. In sostanza containerd-shim permette l'uscita di runc o di un qualsiasi altro runtime container OCI-compliant di uscire dopo che ha avviato il container
4. Runc: Client a riga di comando per mandare in esecuzione containers di formato OCI-compliant

Dal punto di vista della virtualizzazione, Docker engine, come pure qualsiasi altro che si basa sui containers, risiede sopra il sistema operativo host e tutte le applicazioni containers condividono il Kernel del sistema host. I vantaggi dell'architettura container rispetto la VM, è che non vi è extra overhead dovuto all'intero sistema operativo Guest proprio della VM. Il principale vantaggio è una grandissima riduzione della dimensione degli snapshots, tempi di boot e migrazioni del workload più veloci. Figura 10 riepiloga l'architettura software di Docker engine a partire dalla versione 1.11.

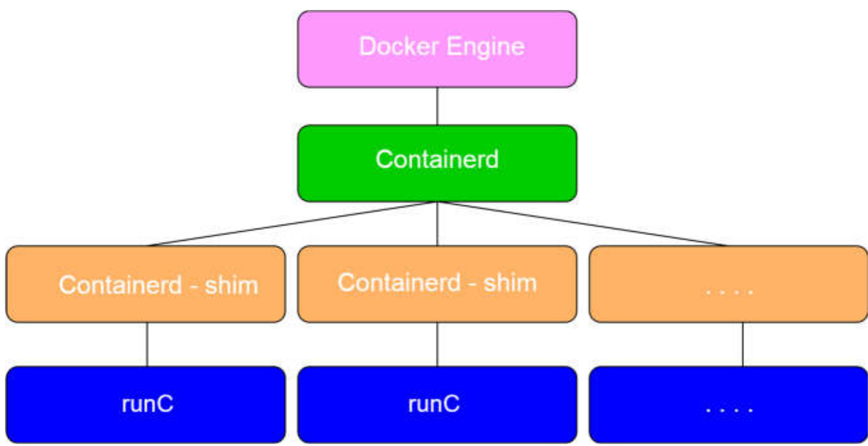

Figura 10 - Docker Engine versione 1.11

Per quanto invece concerne l’aspetto della sicurezza, nonostante i containers siano i candidati naturali per i nuovi servizi MEC, essi continuano a soffrire di potenziali rischi di sicurezza che ad oggi risultato aperti. A causa della condivisione del kernel del sistema host con tutti i containers in esecuzione, una tra le più grandi debolezze di sicurezza è che un attaccante, tramite un programma malevolo, possa scappare dalla sandbox e dirottare il sistema host. Ciò causerebbe la totale compromissione di tutti i servizi gestiti dal kernel dell’host. Figura 11 mostra visivamente tale attacco.

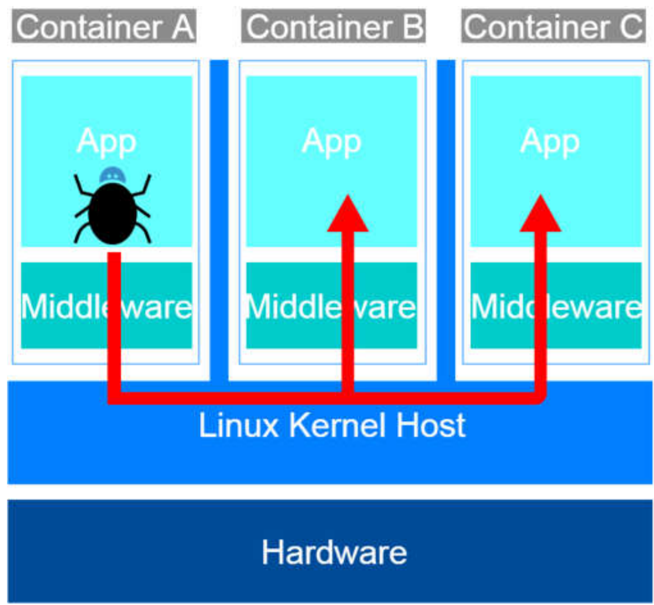

Figura 11 - Problema Sicurezza Docker

1.4 Live Migration

Il MEC permetterà alla rete di prossima generazione di eseguire applicazioni/servizi che richiederanno stringenti condizioni non solo in termini di delay e jitter come i video on-demand ad alta definizione, ma anche applicazioni, quali quelli di realtà aumentata, big data analytics, servizi veicolari, gaming online su dispositivi mobili con limitata potenza di calcolo. Queste applicazioni sono generalmente di tipo client-server, ovvero la parte server, chiamata ‘back-end’ in esecuzione

presso il cloud, mentre la parte client, detta 'front-end' in esecuzione nel dispositivo mobile. Le nuove applicazioni, a causa degli stringenti tempi richiesti come delay della cosiddetta 'tactile-internet' impongono, dunque, la definizione di nuove soluzioni architetturali che avvicinano il cloud agli utenti mobili. Mobile Edge Computing risponderà a tale sfida tramite il deployment di cloud data center, di minori capacità computazionali rispetto a quelli nella core, che saranno direttamente connessi alle base-station cellulari o in siti in prossimità degli utenti lungo tutta la rete di accesso.

Una delle tante sfide introdotte da questa nuova architettura è la *'live migration'* di un'applicazione. Si tratta nello specifico di un server handover oltre che una possibile migrazione della connessione radio dovuta alla mobilità dell'utente. Per potere quantificare le prestazioni di una migrazione di servizio bisogna tener traccia delle pagine di memoria continuamente aggiornate dall'applicazione in esecuzione durante la migrazione, che prendono il nome di *'dirty page'.*

Più in dettagli per 'Live Migration': "*Live Migration è il processo di clonazione di virtual machine (VM) o containers da un server sorgente ad uno di destinazione, mentre il servizio/applicazione rimane in esecuzione durante tale migrazione*".

Come si evince dalla stessa definizione la 'live service migration' si distingue dal service migration dal fatto che in quest'ultima, prima del trasferimento dell'immagine dalla sorgente alla destinazione, il servizio viene sospeso e ripreso a destinazione a trasferimento compiuto. La semplice service migration, a volte chiamata 'cold migration' è molto semplice da implementare, ma causa un lungo periodo di stop dell'applicazione, chiamato 'downtime' che è intollerabile per tutti quei servizi di tipo interattivo. In generale, a prescindere dell'algoritmo considerato, nella live migration si distinguono tre fasi:

1. **Push**: Applicazione in esecuzione presso il server sorgente e contemporaneamente le pagine di memoria sono trasferite a destinazione. Quelle che si sporcano sono ritrasmesse durante il trasferimento fin quando il rate di trasmissione e superiore a quello di sporcamento
2. **Stop and Copy**: Si sospende l'esecuzione della VM/container presso la sorgente e attivata presso il server destinazione dopo il trasferimento delle ultime pagine sporche (dirty pages)
3. **Pull**: Non appena una pagina di memoria, richiesta dall'applicazione in esecuzione presso la destinazione, non è trovata (page fault) si genera una richiesta inviata alla VM/container sorgente lungo la rete di connessione

1.4.1 ALGORITMI DI MIGRAZIONE DATI

A seconda di come sono organizzate le tre fasi, sopra descritte, distinguiamo tre tipi di algoritmi di live migration: '*pre-copy*', '*post-copy*' e '*hybrid-copy*'. In **post-copy**, per prima migriamo una parte minimale della VM come lo stato dei registri della CPU. Dopo che la VM di destinazione riceve e sincronizza tale stato inizia a richiedere le pagine di memoria mancanti tramite network e page fault. Ciò comporta, appunto, un complessivo lungo tempo di downtime causato dall'immediata sospensione del servizio/applicazione. In aggiunta nel post-copy vi sono molti page faults che degradano sensibilmente le prestazioni dell'applicazioni molto sensibili ai ritardi. Riassumendo l'algoritmo di Post-Copy ha un tempo totale di migrazione buono, ma un lungo tempo di downtime che lo rende inadatto per applicazioni RAM-intensive.

Pre-copy, invece è in un certo senso una tecnica inversa poiché iterativamente cerca di migrare il maggior numero di pagine di memoria a destinazione fino a quando non si raggiungono determinate condizioni d'uscita. Distinguiamo due fasi principali:

- Warm-up
- Stop-and-Copy.

Il warm-up è la fase iterativa, consta di 'n' passi, dove le pagine sono duplicate ed inviate a destinazione senza fermare la VM sorgente. In dettaglio, alla prima iterazione tutte le pagine di memoria dell'applicazione sono trasferite dal server sorgente a quello di destinazione senza sospendere l'esecuzione dell'applicazione. Nelle 'n-1' iterazioni se alcune pagine di memoria sono aggiornate durante un trasferimento. Esse saranno ritrasferite nella successiva iterazione fin quando la velocità di (ri)trasferimento delle pagine e non minore della velocità di sporcamento (dirty rate). Nella seconda fase la VM sorgente è fermata e tutte le pagine di memoria rimaste che si sono, nel frattempo, sporcate (chiamate hottest pages) saranno inviate a destinazione insieme al file system (cpu state). Alla fine di questa fase sia presso la sorgente che presso la destinazione vi è un'immagine consistente del servizio/applicazione che sarà messo in esecuzione.

Infine, l'algoritmo di **hybrid-copy** è un mix tra le due tecniche in cui vi è una prima fase di 'push' in cui sono trasferite le pagine di memoria più utilizzate seguita da quella di 'Stop-and-Copy' in cui lo stato della cpu è trasferito (file system) ed infine una fase di 'Pull' qualora dopo l'attivazione del servizio a destinazione vi siano pagine di memoria mancanti. Quest'ultimo algoritmo rispetto a quello di 'Post-Copy' ha prestazioni migliori dovute al minore numero di page faults. La Figura 12 riepiloga graficamente il funzionamento dei tre principali algoritmi.

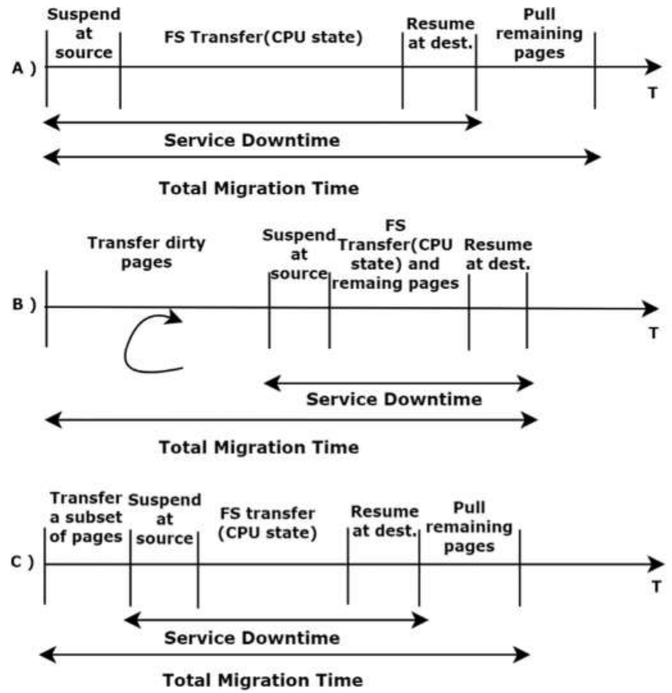

Figura 12 -A) Post-Copy, B) Pre-Copy, C) Hybrid-Copy

1.5 Concetti di Memoria nei Sistemi Operativi Unix

In questo paragrafo introduciamo alcuni concetti chiave quali le metriche impiegate dai sistemi operativi Unix per monitorare il consumo di memoria di un'applicazione in esecuzione. Daremo, dunque, dei cenni sul file system virtuale e su varie metriche che abbiamo poi utilizzato per progettare il nostro classificatore video.

1.5.1 File System Virtuale

Il primo *'file system virtuale'* è chiamato *'/proc'* (16) a volte anche detto pseudo-file system. Esso non contiene file 'reali', ma informazioni di sistema a runtime e può essere considerato come il centro informativo e di controllo per il Kernel. Ogni processo (applicazione in esecuzione) è caratterizzato da un <*PID*> ed il sistema operativo crea per ciascuno di esso un percorso (pathfile) '*/proc/<PID>*' contenente tutti i files ad esso relativi in formato testuale. Molti di questi files sono in solo lettura, ma alcuni di loro hanno accesso in scrittura permettendo, dunque, alcune modifiche a livello Kernel. In particolar modo, le informazioni a livello sistema sull'utilizzo della memoria sono in '*/proc/meminfo*' che fornisce informazioni sull'utilizzo sia fisico che di swap a livello sistema; mentre '*/proc/pid/statm*' fornisce informazioni sullo stato di memoria del processo specificato in <'*PID*'>. Altri due importanti file virtuali per monitorare l'allocazione di memoria per un processo sono: '*/proc/<PID>/map*' e '*/proc/<PID>/smaps*'. Il primo fornisce il mapping dei files eseguibili e delle librerie condivise che compongono il processo nelle regioni di memoria, mentre il secondo mostra il consumo di memoria per ogni mapping del processo. Infine vi è '*proc/<pid>/clear_refs*' che è scrivibile soltanto dal proprietario del processo che può essere utilizzato per resettare tutta una serie di pagine di memoria associate al processo. In più dettaglio andando a resettare i bit '*PG_Referenced*' e '*ACCESSED/YOUNG*' fornisce un metodo di misura in modo approssimativo su quanta memoria un processo sta utilizzando. Infine vi è '*proc/<PID>/pagemap*' che mostra il mapping tra le pagine virtuali e quelle fisiche, dette frames, o aree di swap di un determinato processo.

1.5.2 Metriche Di Memoria

Al fine di poter caratterizzare il processo di occupazione di memoria da parte di una specifica applicazione, in questo lavoro ci affidiamo a diverse metriche di memoria. In generale, ogni applicazione in esecuzione accede a due tipi di memoria: Una '*privata*' che è acceduta soltanto dal processo preso in considerazione ed una '*pubblica*' condivisa da più processi come per esempio le librerie di sistema. La memoria è, in generale, organizzata in pagine che possono essere '*dirty*' o '*clean*'. Dirty significa che le pagine, dopo un certo istante di tempo preso come riferimento, sono state aggiornate a causa per esempio ad una nuova scrittura, mentre clean significano pagine lette, ma non modificate.

Una prima metrica di memoria è *'Virtual Set Size'* (VSS) che rappresenta la quantità totale di memoria virtuale che un processo ha mappato, indipendentemente se è stata o meno scritta in memoria fisica. VSS è una sovrastima poiché le applicazioni normalmente allocano memoria che non usano. '*Resident Set Size*' (RSS) è la quantità di memoria fisica mappata in RAM. Rispetto la precedente è una misura migliore, ma è ancora una sovrastima poiché non tiene in considerazione del numero di processi che utilizzano una o più librerie condivise allocando, dunque, l'intera libreria ad ogni processo che la utilizza. Al fine di fornire una stima più accurata della memoria occupata da un processo vi è la '*Proportional Set Size*' (PSS) che divide la quantità di memoria condivisa per il numero totale di processi che la stanno utilizzando e non come la precedente in cui

la considerava per intero ad ogni processo. Nonostante ciò, anche la PSS, nel momento in cui uno o più processi, che hanno in comune delle librerie condivise, sono arrestati il valore assunto per i rimanenti processi è forviante. Un'altra importante metrica è la '*Unique Set Size*' (USS) che misura la sola memoria privata sia dirty che clean di un dato processo. Questa metrica rappresenta il vero incremento di memoria associato ad un processo e tutte le volte che esso è terminato questa memoria è restituita al sistema. Infine, è molto importante ricordare la definizione di '*working set*' (17) '*Il working set* $W(t, \tau)$*' di un processo al tempo* t *è quell'insieme di informazioni referenziate da un processo durante l'intervallo di tempo* $(t - \tau, t)$ " Esso non è direttamente deducibile dal Kernel, ma nonostante ciò gli attuali sistemi operativi ne fanno uso per decidere una corretta allocazione di memoria nei vari istanti di tempo in cui l'applicazione è in esecuzione.

1.6 DISTRIBUZIONE STATISTICA: GENERALIZED EXTREME VALUE (GEV)

In questo paragrafo introduciamo alcune caratteristiche principali della distribuzione GEV (18). Essa fa parte della famiglia delle distribuzioni continue di probabilità sviluppata entro la teoria dei valori estremi per combinare le famiglie di Gumbel, Fréchet e Weibull anche note come distribuzioni a valori estremi di tipo I, II e III. L' Equazione 1 mostra la funzione di distribuzione cumulativa dove $s = \frac{(x-\mu)}{\sigma}$ è la variabile standardizzata , con $\mu \in \mathbb{R}$ rappresentante il parametro di locazione, $\sigma > 0$ è il parametro di scala e $K \in \mathbb{R}$ è il parametro di forma

$$F(s;K) = \begin{cases} e^{(-(1+Ks))^{\frac{-1}{K}}} & K \neq 0 \\ e^{(-e(-s))} & K = 0 \end{cases}$$

Equazione 1 – Funzione di Distribuzione Cumulativa GEV

L'Equazione 1 per $K > 0$ è valida per $s > {-1}/{K}$, mentre per $K < 0$ è valida per $s < {-1}/{K}$. Più in dettaglio per valori di K positivi la sua funzione densità di probabilità (PDF) ha supporto finito a sinistra e un'infinita coda sulla destra e viceversa per valori di K negativi. Il valore del parametro μ indica dove la PDF è centrata, mentre σ rappresenta l'offset rispetto al valore del parametro μ.

1.7 RIEPILOGO

Questo primo capitolo ha avuto la finalità di introdurre tutta una serie di argomenti propedeutici per potere seguire i lavori svolti durante il corso di Dottorato. In particolare abbiamo dato una panoramica sulle caratteristiche della nuova rete $5G$, successivamente abbiamo introdotto l'architettura di riferimento del MEC e spiegato la differenza tra VM e containers quali tecnologie di supporto per la migrazione dei servizi. Abbiamo, dopo introdotto i principali algoritmi di 'Live Migration' seguita da una panoramica su quali metriche di memoria sono utilizzate dal sistema operativo Unix/Linux per tracciare il loro utilizzo dai vari processi in esecuzione. Infine abbiamo brevemente introdotto la funzione di distribuzione GEV che sarà impiegata in un nostro lavoro.

2 MOBILE EDGE COMPUTING (M-CORD)

Per potere affrontare problematiche riguardanti il Mobile Edge Computing è stato necessario approfondire l'argomento. Abbiamo già introdotto l'architettura generale di riferimento proposta da *'ETSI'* nel primo capitolo. Adesso introduciamo la piattaforma M-CORD (19) che è una soluzione open source per operatori di telefonia mobile che investiranno sul 5G che risulta essere compatibile col modello di riferimento '*ETSI*'.

2.1 PIATTAFORMA M-CORD

Tra le moltissime soluzioni di mobile edge computing, quella che ho ritenuto interessante analizzare è stata M-CORD. Essa fa parte di uno dei tanti progetti il cui capostipite è CORD che sta per '*Central Office Re-architectd as a Datacenter'*. Nasce dall'osservazione che le vecchie centrali telefoniche stavano soffrendo, anno dopo anno, di un elevato aumento del traffico dati. In aggiunta tali centrali sono molto rigide con hardware e software proprietari che rendendo difficile quei requisiti che le nuove infrastrutture devono avere come: integrazione, scalabilità, programmabilità e orchestrabilità. Di conseguenza ci si pose il problema di reingegnerizzare tali centrali e renderle in linea con tali principi. CORD grazie, dunque, all'impiego di tre tecnologie quali NFV, SDV e cloud computing rimpiazza completamente l'hardware chiuso e proprietario con software eseguito su commodity server, switch e dispositivi d'accesso.

M-CORD è appunto uno dei progetti relativamente alla parte mobile. Esso è sia in versione open source che commerciale della core e delle funzioni RAN. Esso abilita servizi innovativi mediante architettura a micro servizi che punta a disgregare le funzioni di rete il più possibile. Fa uso di un modello d'orchestrazione cloud-oriented e le funzioni di rete sono rappresentate come servizi scalabili.

Un grande vantaggio di tale soluzione è che risulta essere in gran parte in relazione con l'architettura di riferimento MEC ETSI introdotta nel capitolo introduttivo. Figura 13 mostra questa relazione tra i moduli MEC ETSI e M-CORD. In particolare abbiamo che il modulo di '*Mobile Edge Orchestrator*' è mappato col modulo '*ONOS*', quello di '*Mobile Edge Platform Manager*' con '*XOS*', quello di '*Mobile Edge Platform*' con '*OPEN VNF*' , il '*data plane*' con '*OpenDayLight*' (ODL) (20), che è un SDN controller, il modulo di 'Virtualization Infrastructure' con 'OpenStack e Kubernetes' (21) (22) , la '*Customer Facing Service Portal*' con l'interfaccia grafica '*Admin GUI (XOS)*'. Infine il modulo di 'Analytic' di CORD non ha alcuna corrispondenza con nessun modulo dell'architettura di riferimento MEC ETSI.

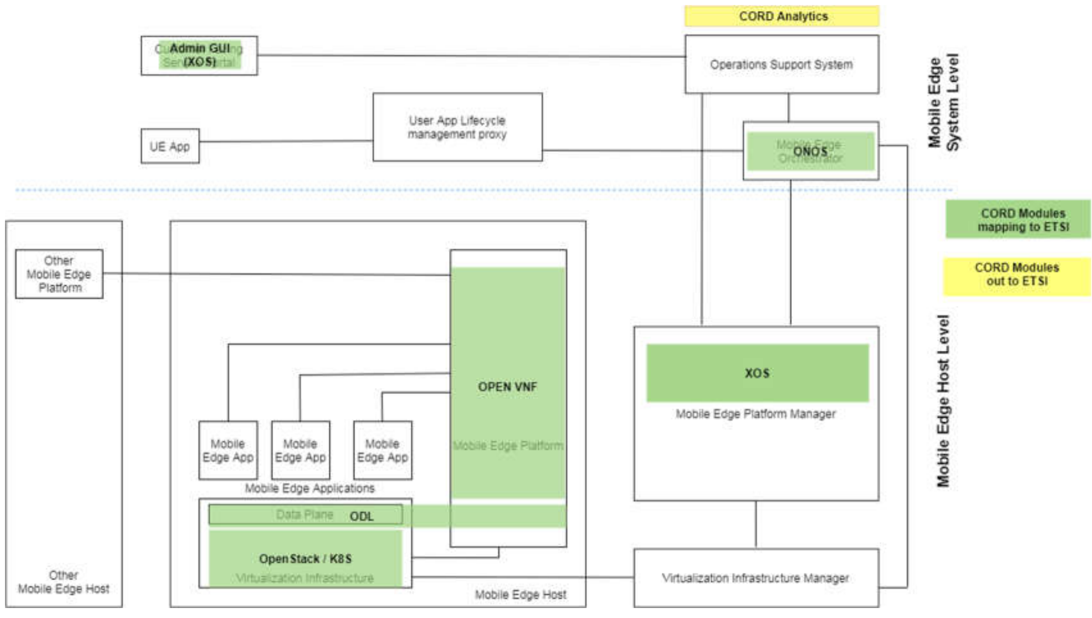

Figura 13 - Relazione tra Architettura MEC ETSI –e CORD

2.1.1 ARCHITETTURA HARDWARE M-CORD

L'architettura hardware di M-CORD e più in generale di CORD è costituita da commodity server e white-box switches, cioè da hardware non più proprietario, tutti disposti in un'unità rack che prende il nome di POD. Il progetto CORD definisce un implementazione di riferimento per i PODs (23), ma è soltanto una delle tante configurazioni che possono essere fatte. Quella di rifermento consiste di 3 principali elementi:

- 6 Servers di tipo OCP (Open Compute Project) configurati con 128 GB di RAM, 2x300 GB HDDs e 40GE dual-port nic
- 6 Switches di tipo OCP e abilitati ad OpenFlow (24) con 32 x 40GE porte
- 2 I/O Blades di tipo OCP con 48 x 2.5 Gb/s interfacce (GPON) e 6 x 40 GE uplinks

Tutti gli elementi sono fisicamente assemblati in un unico rack e virtualmente si hanno due rack come mostrato in Figura 14 mostrante l'architettura hardware CORD.

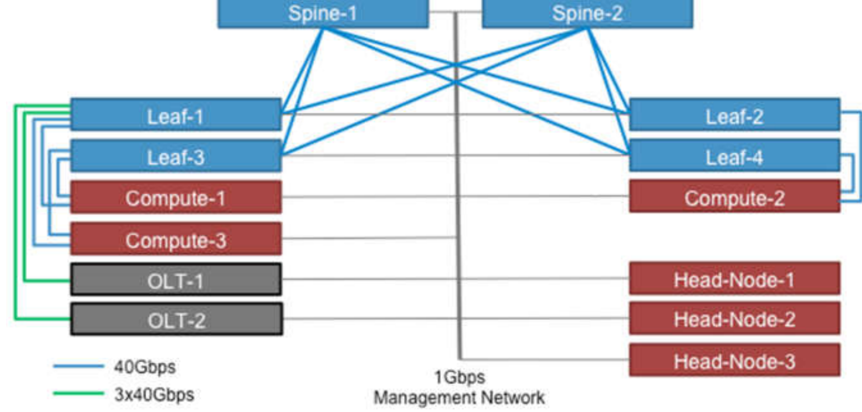

Figura 14 - Architettura Hardware CORD (Fonte: ONF website)

Essi sono interconnessi mediante switches in configurazione "leaf-spine". Per ogni virtual pod vi sono due leaf e due spine switches. Tale configurazione è ideale in tutte quelle situazioni in cui il traffico dati è da est a ovest, ovvero viaggia principalmente dentro il data center e non al di fuori come nella configurazione nord sud. Inoltre permette di aggiungere facilmente addizionale hardware nel momento in cui si ha oversubscription di un link, ovvero quando il traffico generato è maggiore della massima capacità del link, favorendo, dunque, una facile espandibilità. I server eseguono una versione di Ubuntu che include Open vSwitch[1], mentre gli switch eseguono Atrium software stack in cui è presente lo stack di OpenFlow.

2.1.2 Architettura Software CORD

Per quanto riguarda la parte software, l'implementazione di riferimento fa uso di 4 progetti open sources che sono:

1. **Openstack** (21) piattaforma di gestione di cloud datacenter fornendo un servizio di tipo *IaaS*. Ha, inoltre, il compito di creare e installare sia le reti virtuali che virtual machine
2. **Docker** (25) piattaforma di gestione, implementazione dei servizi e loro interconnessione mediante containers
3. **ONOS** (26) è un controller SDN con funzione di sistema operativo di rete allo scopo di gestire sia lo switching software che i sottostanti switches fisici. Esso esegue una serie di applicazioni di controllo e di gestione dello switching così come implementare virtual networks, monitorare links, dispositivi connessi e gestire flussi tra gli switches. In buona sostanza tramite ONOS si ottiene una visione globale della rete software creata e gestita da esso stesso.
4. **XOS** (27) è il framework di riferimento per assemblare e comporre i servizi. Ha il compito di unificare i servizi infrastrutturali di OpenStack con quelli del service plane di ONOS con quelli del data plane e servizi cloud forniti tramite Docker come vitual machine o container. Di fatto XOS realizza la '*service orchestration*'. La Figura 15 mostra l'architettura software CORD dove XOS assembla servizi multi-tenant, ONOS gestisce le reti di switch tramite applicazioni di controllo e OpenStack/Docker/Kubernetes gestiscono le istanze di calcolo.

Figura 15 - Architettura Software CORD (Fonte: ONF website)

[1] Open vSwitch è un'implementazione open-source di switch virtuali multilayer distribuiti. Lo scopo è quello di fornire uno switching stack per ambienti di virtualizzazione hardware per poter supportare diversi protocolli e standard.

2.1.3 XOS FRAMEWORK

Il cuore pulsante dell'architettura CORD è fornita dal framework XOS il quale è strutturato secondo il paradigma del Everything-as-a-Services (XaaS): Ovvero ogni componente è considerato un servizio. Ciò implica che indipendentemente dal fatto che si tratti di un'applicazione di controllo o una virtual machine XOS lo considera come servizio. In buona sostanza fornisce un'astrazione tramite la gestione di *'service controller'* che non sono altro che delle interfacce programmabili. L'implementazione del servizio avviene tramite il service controller che, nascondendo i dettagli implementativi, istanzia un numero scalabile di 'service' che possono essere su cluster geograficamente distanti. La Figura 16 mostra tale paradigma di funzionamento.

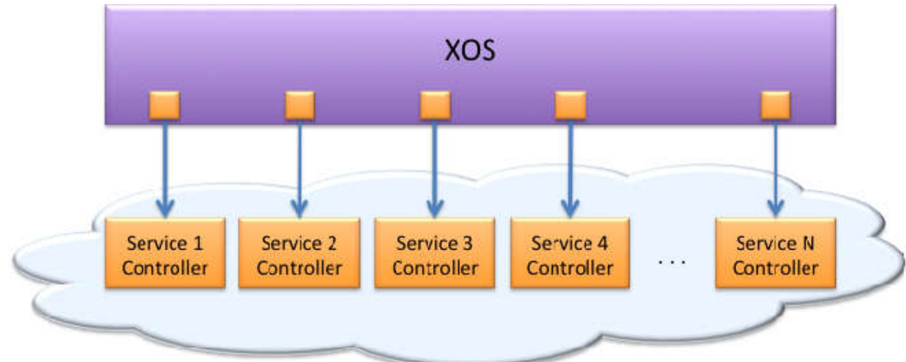

Figura 16 - XOS Controller (Fonte "XoS: An Extensible Cloud Operating System" (88))

Dal punto di vista software XOS è organizzato in 3 livelli: *View*, *Data Model* e *Controller*. Il Data Model è responsabile di unire tutti i servizi tra loro ed ingloba gli oggetti astratti, le loro operazioni e loro relazioni. Tramite l'interfaccia RESTful HTTP o la libreria *xoslib* sono esportate le operazioni degli oggetti per potere svolgere una programmazione di più alto livello tramite le view (viste). Le views sono modellate in funzione dell'utilizzatore se tenant o sviluppatore o operatore. Infine il controller framework ha lo scopo di mantenere lo stato rappresentato dall'insieme dei service controller in sincronia con quello autoritativo del Data Model. Per potere mantenere questo allineamento tra stato operazionale delle risorse e quello autoritativo del Data Model si fa uso di Ansible (28) che sulla base di un playbook, cioè un insieme di azioni da eseguire sulle risorse, eseguirà l'allineamento. In dettaglio è il *Synchronizer che* prima riceve ed analizza il Data Model, poi genera sulla base di un *Model Policy plugin* il playbook che Ansible utilizzerà per sincronizzare le risorse di back-end. La Figura 17 riassume graficamente tale processo di sincronizzazione.

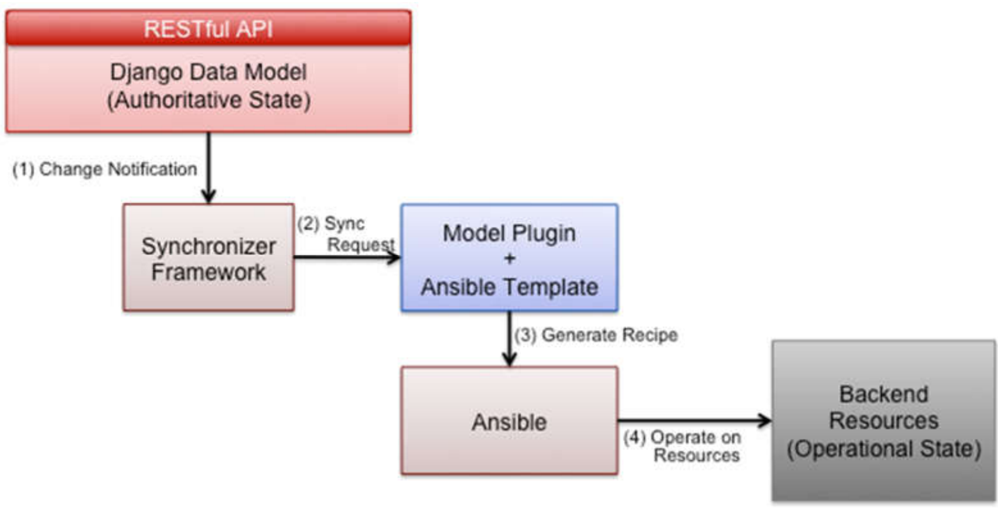

Figura 17 - Service Control Plane (Fonte (85))

Volendo chiarire ulteriormente il ruolo svolto da XOS risulta utile mostrare la Tabella 1 che confronta la visione Unix con quella di XOS. Chiaramente questo mapping non è perfetto, ma serve per comprendere i ruoli giocati dai vari componenti presenti in XOS.

Unix	XOS	Commenti
Processi	Slices(VMs + VNs)	Container di risorse che forniscono isolamento
User, setuid	Tenancy	Controllo d'accesso nel control plane
Shell	Views	Ambiente di programmazione per comporre servizi
Devices	Controllers	Per connettere nuove risorse
Applicazioni	Services	Eseguiti sopra il sistema operativo
Syscall Interface	REST API	Interfaccia a Kernel/Data Model
Libc	Xoslib	Libreria di building block

Tabella 1 - Confronto Unix e XOS

2.1.4 Servizio XOS

Il processo di creazione di un servizio in XOS passa attraverso la scrittura di un Data Model ,delle REST e TOSCA (29) API che servono rispettivamente le prime per interagire col servizio le seconde sono specifiche ad un dato tenant ed un *Synchronizer*. Una volta scritti questi files sarà il processo *Synchronizer che* si occuperà di rilevare ed applicare i cambiamenti del Data Model tramite i playbook di Ansible alle risorse sottostanti. Infine per ogni servizio vi è un *on-boarding playbook* che ha la funzione di caricarlo su XOS ed attivarlo. Tale file è scritto in TOSCA e contiene tutti i path ai file creati precedentemente che serviranno al funzionamento del servizio. XOS fornisce un'interfaccia grafica raggiungibile via browser, di nome *Service Graph* per vedere tutti i servizi attivi e caricati insieme alle loro dipendenze. Nel caso di M-CORD, che rappresenta uno dei possibili profili, bisogna mapparlo sul Service Graph. In buona sostanza un servizio in XOS non è altro che '*un oggetto astratto costituito da un insieme di risorse, chiamate slice, ed un controller.'* Per sintetizzare gli oggetti chiave che costituiscono un servizio sono:

1. Service = ({Slice,….}, Controller)
2. Slice = ({VM,…},{VN,…})
3. VM = (Collocazione, Immagine, Risorse)
4. VN = (Topologia, Sistema Operativo di Rete (ONOS), Risorse)
5. Controller = (URL, Credenziali, Plugin)

2.1.5 Composizione Di Un Servizio XOS

Dato che XOS è un sistema modulare, la sua forza sta nel poter comporre questi servizi tra loro per crearne di nuovi e più complessi. La composizione avviene mediante due o più servizi di cui uno è il tenant mentre l'altro è il provider. Dunque, nel data model, questa relazione è rappresentata come un *'Tenant Object'* che è costituito oltre che dai due servizi da un component *'Connect'* che specifica come connettere i due servizi nella rete sottostante ed *'Attributes'* che registra lo stato necessario per implementare questa 'tenancy':

Tenant = (Service_T, Service_P, Connect, Attributes)

Bisogna osservare che a differenza dei servizi cloud multi-tenancy in XOS ci sono due importanti differenze che necessitano essere indirizzate per potere comporre un servizio. La prima è che un tenant non identifica un utente in XOS, ma è esso stesso un servizio. Ciò significa che se un tenant è un servizio scalabile tutte le sue istanze devono poter accedere al servizio provider. La seconda è che tali servizi devono essere autonomi. Sebbene queste due sfide possano essere indirizzate in fase di definizione del servizio stesso XOS fornisce meccanismi che riducono il costo operazionale che un servizio A sia un tenant di un servizio B. Questi meccanismi si traducono nell'abilitare la composizione nel data plane e nel control plane. XOS fornisce tre modalità per interconnettere nel data plane una coppia di servizi:

1. Comunicazione tramite rete pubblica Internet
2. Creare una virtual network condivisa tra due o più slices che interconnette tutte le VMs appartenenti alle varie slices partecipanti
3. Tramite OpenVirteX (30)

Altro aspetto della service composition è relativa al control plane che significa gestire la tenancy di un servizio rispetto ad un altro. Questo avviene memorizzando lo stato di tenancy nel data model che corrisponde a specificare che A è un tenant di B tramite il campo 'Attributes' dell'oggetto tenant. Inoltre le varie dipendenze presenti tra i servizi sono registrate anche nel data model ed in corrispondenza di un cambiamento di stato XOS lo notifica tramite il controller.

2.1.6 Mapping M-CORD su XOS

Per poter mappare M-CORD su XOS bisogna prima di tutto chiarire il significato di xslice e mslice.

- Xslice *'risorse allocate ad un servizio'*
- Mslice *'risorse allocate ad un insieme di servizi ad una classe di utenti'(Insieme di XSlice)*

Per dato profilo M-CORD bisogna mapparlo sul service graph che rappresenta l'insieme delle funzionalità che vogliamo considerare, poi ogni servizio sarà mappato su un insieme di risorse (XSlice) al fine di poter soddisfare la QoS per dato profilo. La Figura 18 - M-CORD mapping on XOS mostra il mapping di M-CORD su XOS.

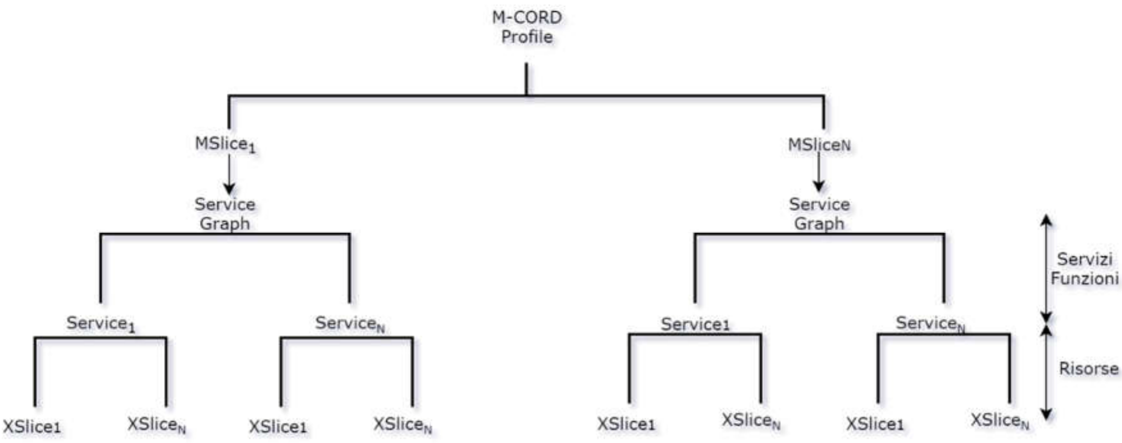

Figura 18 - M-CORD mapping on XOS

Le funzioni sia radio che quelle della core sono virtualizzate come servizi mobili e gestiti in modo elastico e scalabile dal proprio tentant. XOS orchestra queste funzioni NFV. Le principali funzioni virtualizzate sono:

- BBU (Baseband Unit)
- MME (Mobility Management Entity)
- SPGW (Service Packet Gateway sia parte di controllo che quello dati)
- HSS (Home Subscriber Server)

La Figura 19 mostra come XOS realizza un insieme di servizi per un'architettura di rete mobile virtualizzata.

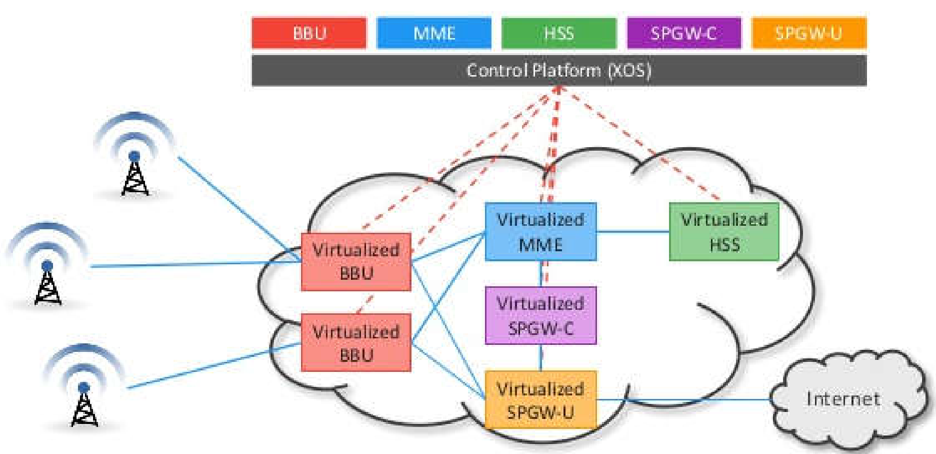

Figura 19 - Architettura dei Servizi Mobili Virtualizzati

Questi servizi devono essere composti e posti in relazione di dipendenza tra loro cioè bisogna creare una 'Service Chain'. La Figura 20 mostra un esempio di '*mobile service chain*' nel caso in cui si voglia fornire un servizio di telefonia mobile. Bisogna notare che le frecce non indicano il flusso dei dati, ma il rapporto di dipendenza tra i servizi composti.

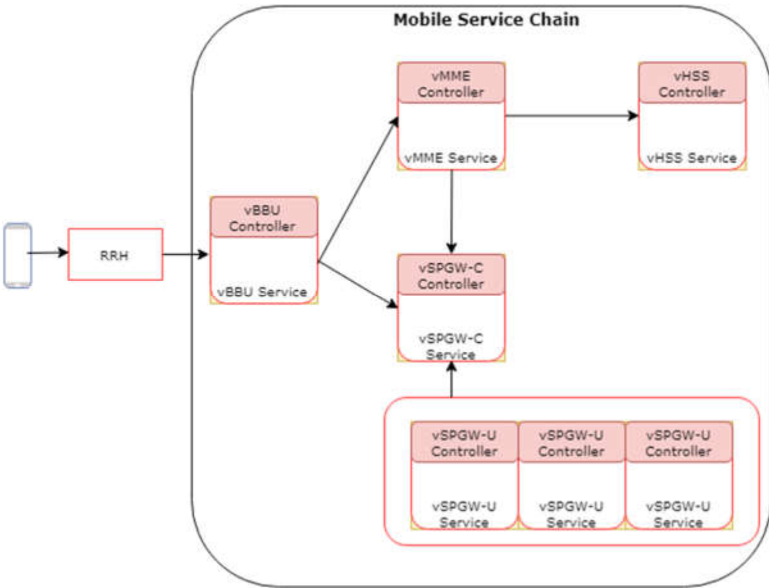

Figura 20 - Mobile Service Chain

Per ogni servizio, il corrispondente controller, a secondo del caso d'uso che sarà eseguito per una certa classe di utenti, implementerà un determinato numero di istanze in modo totalmente dinamico, scalabile, altamente personalizzabile e con diversi livelli di qualità del servizio raggiungendo un'agilità e riduzione dei costi nel rilascio dei servizi mai visto sino ad oggi nelle reti mobili.

2.2 Riepilogo

In questo capitolo abbiamo introdotto in dettaglio una nuova e promettente soluzione per abilitare futuri servizi 5G tramite Mobile Edge Computing. La piattaforma analizzata è stata M-CORD che rappresenta un particolare caso d'uso di un progetto open source di nome CORD che punta a mettere insieme tre principali tecnologie: SDN, NFV e cloud. Grazie a tale soluzione avremo un miglioramento dell'utilizzo in tempo reale delle risorse, analisi dati in tempo reale, sfruttamento di multiple tecnologie d'accesso radio. Sarà, inoltre, possibile un'elevata personalizzazione dei servizi con differenti valori di QoS (Quality of Service) grazie alla loro composizione dando la possibilità di abilitare differenti casi d'uso come IoT (Internet of Things), smart cities, applicazioni M2M (Machine to Machine), istruzione, ambito sanitario e tanti altri. Infine non meno importante le installazioni diventeranno on-demand, grazie alla '*softwarizzazione della rete*' disaggregando le funzionalità sia della RAN che della core EPC. Inoltre la possibilità di installare tale piattaforma su commodity server e non più su un costoso hardware chiuso e proprietario permetterà un abbattimento dei costi di rilascio, installazione e gestione.

3 ANALISI PRESTAZIONALI DI LIVE MIGRATION DI APPLICAZIONI MEMORY-INTENSIVE IN MEC

Questo capitolo descrive un primo lavoro su una delle tante sfide che vi sono in Mobile Edge Computing. In particolare affronteremo il problema della migrazione dei servizi tra due MEC a causa non solo della mobilità utente, ma soprattutto per la mancanza di risorse necessarie presso il source MEC per mantenere il servizio attivo. Questo lavoro è un'analisi di un particolare tipo di applicazione, detta memory-intensive, cioè applicazioni per cui la quantità di memoria principale (RAM) utilizzata è molto maggiore rispetto alla quantità di spazio occupata su memoria secondaria della cosiddetta 'footprint' di installazione. Il capitolo sarà così strutturato: Un'introduzione in cui si fa il punto sulle precedenti ricerche nel campo della service migration. Successivamente s'introduce un'architettura per la 'Service Migration' che va a migliorare i tempi di migrazione. Dopo introdurremo, sotto determinate ipotesi, un modello di migrazione basato sull'algoritmo di 'pre-copy' e ne analizzeremo le prestazioni in termini di tempo totale di migrazione che tempo di 'downtime'. Infine chiuderemo il capitolo con un riepilogo.

3.1 INTRODUZIONE

Come descritto nel precedente capitolo, MEC permetterà alla rete di prossima generazione di eseguire applicazioni/servizi che richiederanno stringenti condizioni non solo in termini di delay e jitter come i video on-demand ad alta definizione, ma anche applicazioni, quali quelli di realtà aumentata, big data analytics, servizi veicolari, gaming online su dispositivi mobili con limitata potenza di calcolo. Queste applicazioni sono generalmente di tipo client-server, ovvero la parte server, chiamata *'back-end'* in esecuzione presso il cloud, mentre la parte client, detta *'front-end'* in esecuzione nel dispositivo mobile. Le nuove applicazioni, a causa degli stringenti tempi richiesti come delay della cosiddetta *'tactile-internet'* impongono, dunque, la definizione di nuove soluzioni architetturali che avvicinano il cloud agli utenti mobili. Mobile Edge Computing risponderà a tale sfida tramite il deployment di cloud data center, di minori capacità computazionali rispetto a quelli nella core, che saranno direttamente connessi alle base-station cellulari o in siti in prossimità degli utenti lungo tutta la rete di accesso.

Una delle tante sfide introdotte da questa nuova architettura è la *'live migration'* di un'applicazione. Si tratta nello specifico di un server handover oltre che una possibile migrazione della connessione radio dovuta alla mobilità dell'utente. Per potere quantificare le prestazioni di una migrazione di servizio bisogna tener traccia delle pagine di memoria continuamente aggiornate dall'applicazione in esecuzione durante la migrazione, che prendono il nome di *'dirty page'*. In letteratura troviamo diversi lavori di cui vale la pena citarne qualcuno. Il lavoro (31) propone uno schema ottimizzato di trasferimento delle dirty page che si aggiornano più frequentemente basato su hypervisor XEN (32). Il lavoro (33) propone un'altra soluzione di ottimizzazione dove insieme all'algoritmo di pre-copy viene abbinato un estimatore capace di fornire in anticipo le pagine con più alta probabilità di essere aggiornate e, dunque, più indicate ad essere trasferite solo nella fase finale dell'algoritmo al fine di ridurre ritrasmissioni delle stesse pagine. Vi sono, ancora, altri lavori in cui si punta a tecniche di compressione per ridurre la quantità di dati da migrare. Un interessante lavoro dal quale si è preso spunto per il nostro lavoro è stato quello di (34). In tale lavoro si propone di decomporre virtualmente una generica applicazione in tre livelli: *base layer*, *application layer* e *instance layer*.

Tramite tale suddivisione a livelli sarebbe possibile piazzare alcuni di essi presso il target MEC così da ridurre nel complesso i tempi di migrazione di downtime. A partire da questo framework a livelli, abbiamo definito un nuovo modello di migrazione ottimizzato basato sull'algoritmo di pre-

copy e proposto un meccanismo per quantificare i tempi di migrazione e downtime in funzione di ipotetiche applicazione *RAM-intensive*, ossia la cui frequenza di aggiornamento/sporcamento delle pagine di memoria è elevata ed inoltre la quantità di RAM coinvolta è maggiore rispetto lo spazio occupato nella memoria secondaria dalla sua stessa installazione. Questo lavoro di ricerca è stato presentato alla conferenza di Valencia '*The Fourth IEEE International Workshop on Mobile Cloud Computing systems, Management, and Security (MCSMS-2018)'* (35)

3.2 Architettura a Tre Livelli per la Service Migration

Allo scopo di ottimizzare la migrazione di un servizio è stato introdotto un modello architetturale in cui virtualmente s'immagina un servizio decomponibile in tre parti (34). In particolare un servizio è visto come la combinazione di un '*livello base'*, che è l'immagine del sistema operativo col suo kernel, un *'livello applicazione'*, che rappresenta l'applicazione in stato 'idle' (di fatto il footprint della sua installazione) ed un terzo *'livello istanza'*, che rappresenta l'applicazione in stato 'running'. La Figura 21 mostra questo modello architetturale:

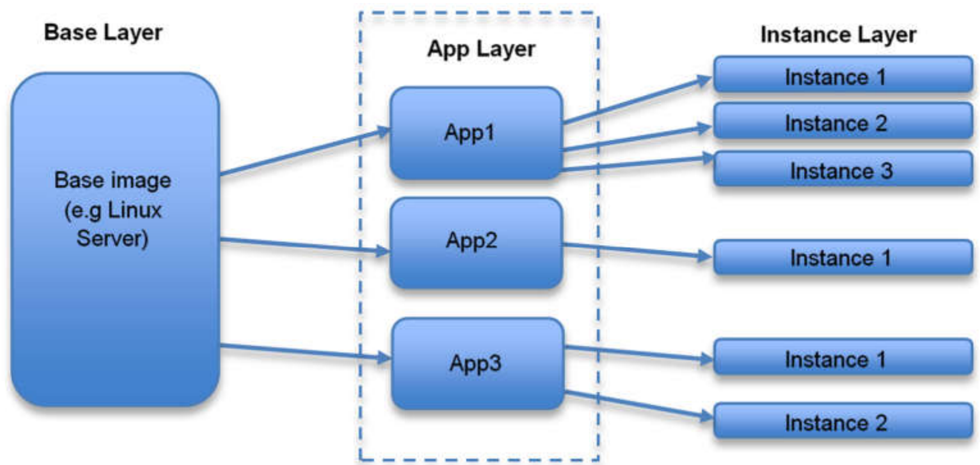

Figura 21 - Architettura a tre Livelli

Questa decomposizione permette di clonare un livello inferiore (per esempio il livello applicativo) prima di eseguire una sincronizzazione tra source e target server a livello superiore (per esempio a livello d'istanza). Tale architettura avvalendosi fondamentalmente di due primitive disponibili nei sistemi server 'sincronizzazione' e 'clonazione' applicata sia a livello applicativo che a quello d'istanza riesce a fornire prestazioni soddisfacenti in termini di downtime per buona parte delle tipologie di servizio da migrare. La ragione di tale risultato sta nell'avere introdotto proprio questo livello intermedio. Infatti tramite questo livello intermedio la maggior parte dei dati dell'applicazione sono trasferiti prima di sospenderla presso il server sorgente. La Figura 22 chiarisce quanto esposto da un punto di vista temporale quando avviene la migrazione di un servizio. Infatti nell'ipotesi in cui a destinazione sia già presente una copia dell'applicazione in stato 'idle' (fase indicata col numero 4) basterà in fase di migrazione sincronizzare solo lo stato della memoria principale e dei registri cpu che si sono modificati, anziché l'intero footprint dovuto all'installazione dell'applicazione presso la destinazione, tutto ciò si tradurrà in un minore downtime, che ricordiamo iniziare nel momento in cui la sorgente è sospesa (fase indicata col

numero 5.x). Quindi, il grande vantaggio di suddividere logicamente un'applicazione in (base, applicazione, istanza) permette, sotto le ipotesi sopra scritte, di ridurre i tempi di downtime e più in generale quelli di migrazione.

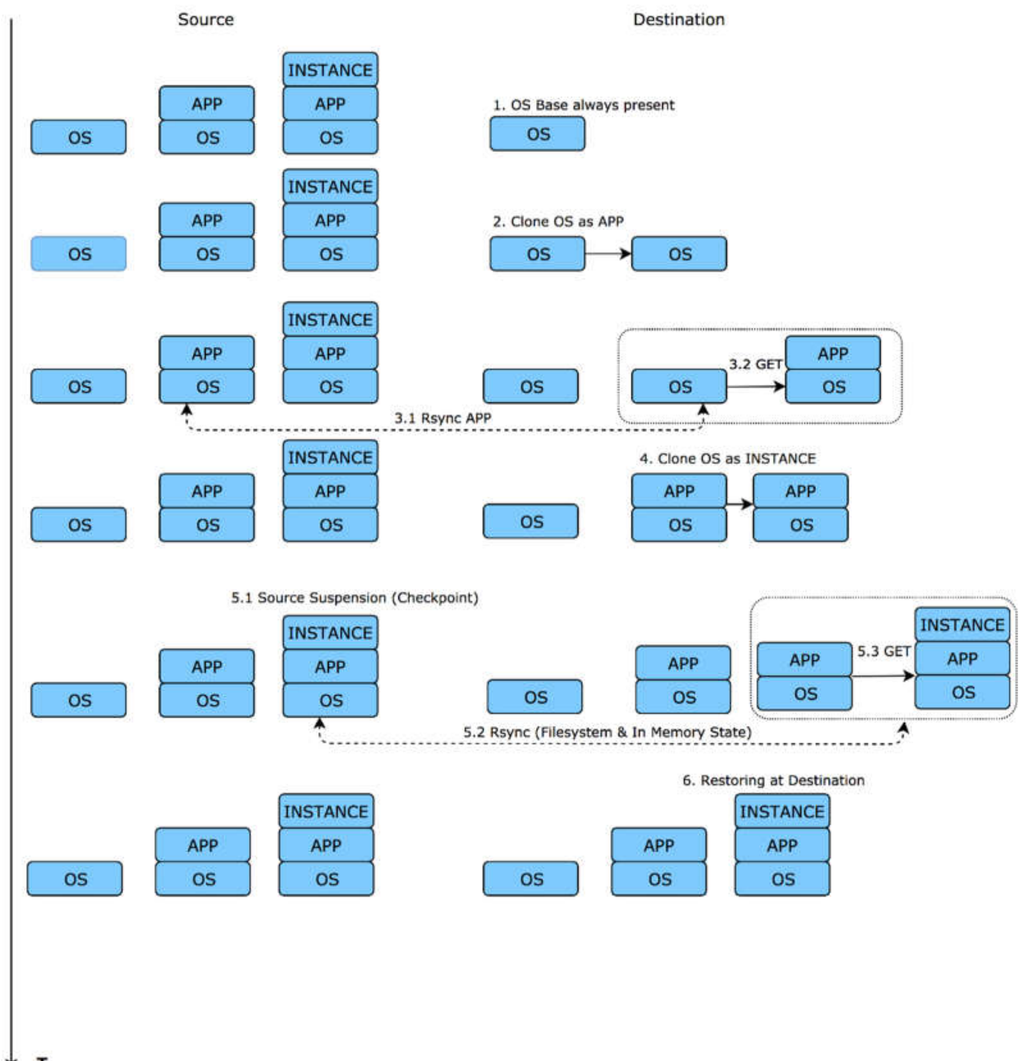

Figura 22 - Clonazione e Sincronizzazione dei dati

3.3 Architettura a Tre Livelli Modificata

Sebbene l'architettura a tre livelli avesse mostrato buoni risultati per una larga scelta di tipologie di servizio in termini di downtime, lo stesso non si è verificato per applicazione di tipo *RAM-Intensive*. Le cause di un tale calo prestazionale sono da ricercarsi in alcune scelte fatte nel precedente modello ed in particolare non aver tenuto conto delle caratteristiche delle applicazioni RAM-Intensive. I principali limiti che abbiamo notato e che ci hanno permesso di definire un nuovo modello di migrazione dei servizi ottimizzato per tali applicazioni sono stati:

1. Utilizzo algoritmo di Post-Copy
2. Pre-Assenza del livello applicativo nel server destinazione
3. Utilizzo della tecnologia di Virtual Machine per migrazione dei servizi

L’algoritmo di post-copy, per definizione, trasferisce le pagine di memoria dopo che il server sorgente è sospeso. Poiché la quantità di memoria coinvolta per le applicazioni RAM-Intensive può essere dell’ordine di centinaia di mega bytes si deduce che il tempo di downtime può risultare estremamente alto perfino per trasferire il solo stato della memoria e dunque intollerabile per tutte le applicazioni che richiedono stringenti delay. Secondo, l’escludere la contemporanea presenza presso il server destinazione sia del livello base che del livello applicativo introduce inevitabilmente ulteriori tempi di clonazione e successive sincronizzazioni che si vanno ad aggiungere al tempo totale di trasferimento. Terzo, per i motivi precedentemente esposti, se pur vero che le virtual machine sono una tecnologia ben consolidata non offrono quella ‘leggerezza’ e ‘velocità’ che saranno richieste per molti tipi di applicazioni, contribuendo, dunque, ad aumentare i tempi di migrazione a causa del loro maggiore overhead. Per tali ragioni ho proposto la seguente architettura a tre livelli modificata:

1. Utilizzo algoritmo di Pre-Copy
2. Presenza anche del livello applicativo presso ogni server di destinazione
3. Utilizzo della tecnologia dei Containers per la migrazione dei servizi

Queste modifiche consentirebbero, anche ad applicazioni quali appunto le RAM-Intensive, una live migration senza particolari problemi. Infatti rimpiazzando il tipo di algoritmo avremmo il vantaggio di migrare le pagine di memoria durante la fase iterativa cercando di ridurre al minimo le pagine di memoria da trasferire solo dopo la sospensione del server sorgente. In particolare le pagine rimanenti sarebbero quelle la cui frequenza di sporcamento è talmente elevata rispetto alla velocità del link che necessariamente bisognerà trasferirle alla fine della migrazione. Così facendo una buona parte dell’instance layer sarà già stato trasferito con un conseguente minore downtime. Altro miglioramento, anche se a scapito di maggiore uso di storage presso i server destinazione, è la presenza, oltre che dell’immagine base, anche di quella applicativa che come già sottolineato va ad eliminare i tempi richiesti sia di clonazione e successiva sincronizzazione. Infine il rimpiazzamento delle virtual machine con i containers porterebbe maggiore reattività e minore latenza durante il trasferimento.

3.4 Analisi dei Tempi di Migrazione per RAM-Intensive Application

Sulla base delle precedenti assunzioni abbiamo condotto un'analisi sulle prestazioni dei tempi di migrazione di un servizio ipotizzando applicazioni RAM-Intensive in un sistema avente XEN come hypervisor basato sull'algoritmo di pre-copy. Abbiamo, dunque, creato tramite Matlab, un modello che emulasse la migrazione del servizio. Tra le ipotesi, senza perdere di generalità, abbiamo supposto che la memoria secondaria occupata dall'applicazione dopo la sua installazione fosse trascurabile rispetto a quella occupata in memoria principale. Così facendo, trascuriamo tutti i relativi tempi di inizializzazione, resuming, ed attivazione nel valutare sia i tempi di downtime che quelli di migrazione. Possiamo, dunque, approssimare, per applicazioni RAM-Intensive, il tempo totale di migrazione come somma di due principali contributi pari rispettivamente alla fase di pre-copy (detto anche warm-up time o di migrazione) più quella di stop-and-copy. Facendo sì che il tempo totale di migrazione risulti pari a:

$$T_{TotalMigration} = \sum_{i=1}^{n} T_{rsync}(i) + T_{rsync_stop}$$

Dove n è il numero di iterazioni durante la fase di warm-up, $T_{rsync}(i)$ è il tempo di risincronizzazione richiesto per trasmettere tutte le pagine di memorie che si sono sporcate alla fine della i-esima iterazione e T_{rsync_stop} rappresenta il tempo necessario per trasmettere le pagine di memoria dopo che l'applicazione è fermata presso il server sorgente, ovvero il nostro downtime. Sia il tempo di warm-up che quello di downtime dipendono da diversi parametri di sistema quali velocità del link (larghezza di banda) di connessione tra sorgente e destinazione, totale memoria RAM occupata dall'applicazione espressa in pagine e la velocità media di aggiornamento (sporcamento) alla quale le pagine di memoria sono aggiornate.

Identifichiamo con R_{avg} la velocità media di aggiornamento delle pagine espressa in pagina/secondo e con M il numero totale di pagine di memoria dell'applicazione. Nell'ipotesi che la probabilità $p(i)$ che una data pagina di memoria risulti sporca dopo una generica i-esima iterazione sia costante e statisticamente indipendente da tutte le altre pagine. Allora il numero di pagine che risultano sporche dopo una generica iterazione i-esima che dura $T_{rsync}(i)$ seguono una distribuzione binomiale con parametro $p(i) = \frac{R_{avg} * T_{rsync}(i)}{M}$. Assumendo che ogni pagina di memoria ha un tempo di trasferimento costante T_{page}, la durata di ogni iterazione dipende dalle pagine residue che si sono sporcate nell'iterazione precedente , cioè $T_{rsync}(i+1)$ è funzione di $T_{rsync}(i)$ per mezzo della $p(i)$:

$$T_{rsync}(i+1) = \sum_{x=0}^{M} \binom{M}{x} p(i)^x (1-p(i))^{M-x} * T_{page}$$

Da ciò consegue che il tempo totale di migrazione può essere calcolato partendo dalla prima iterazione in cui tutte le pagine di memoria sono migrate a destinazione ($T_{rsync}(1) = M * T_{page}$).

Siccome l'hypervisor XEN si basa sull'algoritmo di pre-copy, il numero di iterazioni da svolgere non è noto a priori, ma limitato superiormente da una soglia di $Massimo\ Numero\ di\ Iterazioni = 29$. Tale soglia serve per evitare che la fase di warm-up sia troppo lunga, per esempio a causa di

una velocità di sporcamento delle pagine maggiore rispetto a quella offerta dal link. Inoltre sono stabiliti altri due parametri, sempre allo scopo di uscire opportunamente dalla fase iterativa, che sono: '*numero di pagine residue da migrare*' pari a $N = 50$ e un parametro $K = 3$ che serve a controllare se la somma complessiva delle pagine di memoria trasmesse e ritrasmesse durante la fase di warm-up supera un valore pari a K volte il numero di pagine di memoria di cui consta l'applicazione. Quindi per ricapitolare l'algoritmo di pre-copy ad ogni iterazione verificherà che il numero di iterazioni svolte è minore della soglia, che il numero di pagine trasferite sino a questo momento sono inferiori a 3 volte il numero totale di pagine dell'applicazione e che le pagine residue siano maggiori di 50. Se queste condizioni sono tutte e tre verificate vi sarà una successiva iterazione viceversa si esce dalla fase di warm-up e si calcolerà il downtime come rapporto delle pagine di memoria residue e la velocità del link.

3.5 Analisi dei Risultati Numerici

Come scritto prima abbiamo implementato e simulato in MATLAB un sistema funzionante secondo il modello descritto nel precedente paragrafo per supposte applicazioni RAM-Intensive. Piuttosto che valutare i tempi medi di migrazione, abbiamo eseguito alcuni processi andando a variare sia numero totale di pagine e velocità del link di collegamento. Quindi per ogni dato R_{avg} avremo svolto ,alla fine di ogni simulazione, un certo numero di iterazioni di durata $T_{rsync}(i)$ la cui somma rappresenterà il nostro tempo di warm-up e un tempo di downtime pari al rapporto delle pagine di memoria residue e la velocità del link. In particolare, abbiamo considerato applicazioni con un occupazione di memoria RAM da $64\ MB$e $256\ MB$, mentre per la velocità dei link abbiamo considerato rispettivamente : $100\ Mbps$, $1\ Gbps\ e\ 10\ Gbps$ tra sorgente e destinazione e assumendo una pagina di memoria di dimensione pari a $4\ KB$. Il codice sorgente principale con relativa tabella che spiega le variabili coinvolte è presente in APPENDICE A. Una prima analisi è stata quella di valutare il service downtime raggiunto in un'istanza random del nostro processo di migrazione, come funzione di R_{avg} per un'applicazione di $64\ MB$, che equivale a 15625 pagine di memoria. Abbiamo analizzato i tempi considerando tutti e tre i link di collegamento. Osservando la curva in Figura 23 relativamente ad un collegamento di $100\ Mbps$ possiamo notare due regioni :

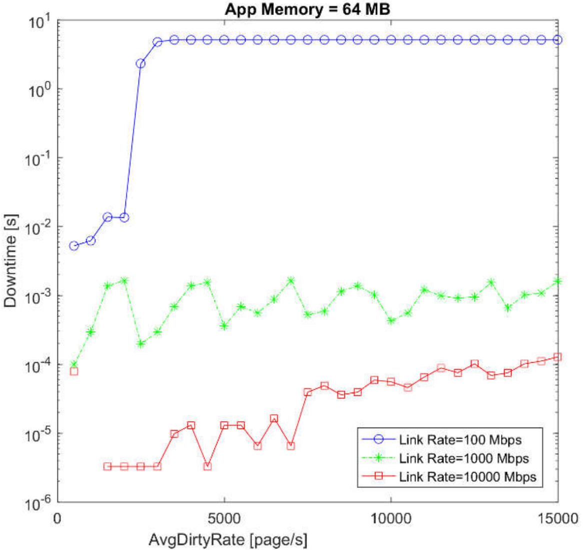

Figura 23 - Service Downtime Analysis

La prima, quando R_{avg} è $< {}^{1}/_{T_{page}}$ il service downtime aumenta linearmente con il dirty rate (Regione Lineare) , mentre quando R_{avg} è $> {}^{1}/_{T_{page}}$ il service downtime raggiunge un valore costante (Regione di Saturazione). Inoltre, non appena il dirty rate supera la velocità di trasferimento del collegamento, il numero finale di pagine di memoria rimaste da trasferire durante la fase di *stop-and-copy* è praticamente pari al numero totale di pagine costituenti l'applicazione e cioè pari a M, il che produce il massimo service downtime pari $M * T_{page}$. Dalla Figura 23 notiamo anche che all'aumentare della velocità del collegamento, le condizioni di saturazione potrebbero non essere mai raggiunte; in tali circostanze , il service downtime è basso grazie alla proporzionale diminuzione del valore di T_{page}

Come già osservato, le fluttuazioni presenti in regione lineare al variare di R_{avg} sono dovute al fatto che i risultati si riferiscono a singole istanze di simulazione e non ad un valore medio di N simulazioni per dato R_{avg}. Da notare che le condizioni d'uscita, prima di raggiungere la fase finale di stop-and-copy, possono essere raggiunte dopo un numero variabile di iterazioni. In particolare, nella regione lineare quando la condizione d'uscita è che il numero di pagine residue è minore della soglia $MAX_{left_page} = 50$ otteniamo che il numero di iterazioni è una funzione non decrescente del R_{avg}. Per un dato numero di iterazioni i, il valore medio delle pagine residue $E[X(i)]$ da trasferire nella fase di stop-and-copy aumenterà con l'aumentare del R_{avg} . Siccome il service downtime è proporzionale ad $E[X(i)]$ ne consegue che a parità di iterazioni il service downtime aumenterà. Tale comportamento è evidenziato in Figura 24 che mostra il numero di iterazioni compiute prima che si verifichi una delle condizioni d'uscita e il corrispondente service downtime per un link avente una velocità di 1 $Gbps$.

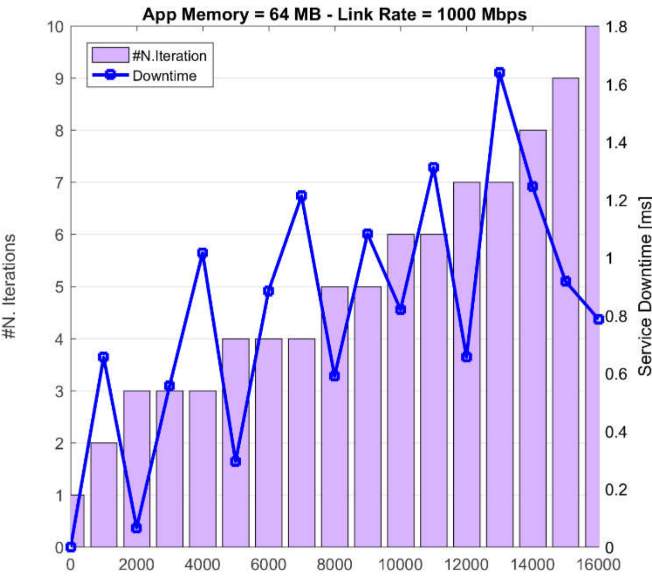

Figura 24 - Numero di Iterazioni e Service Downtime al variare di Ravg

Sempre osservando la Figura 24 si nota un andamento del service downtime a *'dente di sega'* che mostra un incremento lineare a parità di numero di iterazioni $'i'$ dovuto all'incremento delle pagine residue $X(i)$ ed una brusca discesa non appena il numero di iterazioni incrementa da $'i'$ a $'j'$ con $j > i$. Questo comportamento è conseguenza del fatto che, pur aumentando la velocità di sporcamento medio delle pagine R_{avg} è altrettanto vero che grazie ad un ulteriore iterazione si ha la possibilità,

prima della fase di stop-and-copy, di trasferire ulteriori pagine di memoria risultando dunque in un decremento delle pagine residue $X(j)$ e di conseguenza anche del service downtime.

Infine, la Figura 25 mostra il tempo totale di migrazione (figura nella parte superiore) e quello di service downtime (figura nella parte inferiore) nell'ipotesi di un'applicazione RAM-Intensive con un numero di pagine pari a $M = 62500$ equivalente a $(256\ MB)$ su un link a $10\ Gbps$ al variare della soglia della condizione uscità 'MAX_{left_page}' La Figura 25 mostra chiaramente come il service downtime sia fortemente influenzato dalla variazione di tale soglia, mentre risulta quasi invariato il tempo totale di migrazione dovuto principalmente al fatto che il contributo del service downtime rispetto a quello di warm-up risulta trascurabile.

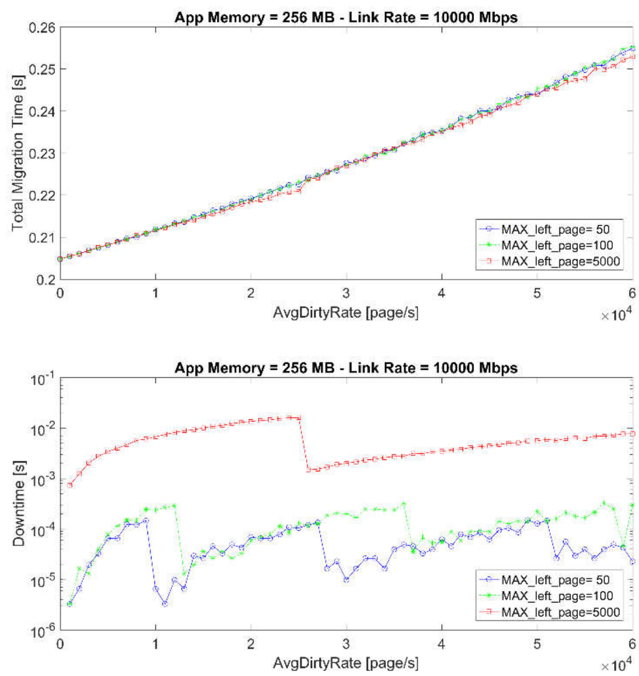

Figura 25 - Tempi di Migrazione al variare di MAX_left_page

3.6 Riepilogo

Per mezzo di tale lavoro di ricerca abbiamo descritto differenti soluzioni per minimizzare il service downtime durante la migrazione di un'applicazione nei futuri scenari del Mobile Edge Computing. Abbiamo considerato un promettente approccio a livelli per decomporre applicazioni che possono essere presenti presso i mobile edge server destinazione andando, quindi, a limitare la fase di clonazione e sincronizzazione. Abbiamo, inoltre, presentato un'analisi per quantificare i tempi di migrazione ed in particolare il service downtime come funzione dei principali parametri del sistema. Da tale analisi emerge che l'algoritmo di pre-copy, grazie alla sua fase iterativa di warm-up, può essere un buon approccio per ridurre il service downtime prima di eseguire la fase di stop-and-copy. Da tale studio sono emerse due considerazioni per future analisi. La prima è che tramite un'ottimizzazione dei parametri d'uscita dell'algoritmo la fase di pre-copy potrebbe essere migliorata visto la forte influenza che hanno tali parametri sul service downtime. Secondo, la velocità del link non è in realtà costante, ma dipende dal traffico di rete in quel momento e che quindi tramite un SDN controller avente istante per istante le condizioni di traffico dei vari link colleganti sorgente e destinazione potrebbe ottimizzare le prestazioni dei tempi di migrazione.

Una possibile estensione del lavoro sarà quella di introdurre dei modelli di traffico di rete per potere simulare una velocità, ossia larghezza di banda variabile ed introdurre un controller che venga in ausilio all'algoritmo di pre-copy per migliorare i tempi di migrazione.

4 RICONOSCIMENTO RISOLUZIONE VIDEO VIA METRICHE DI MEMORIA

Content Delivery sarà uno dei casi d'uso di maggior popolarità in ambito 5G MEC visto che il numero di abbonati ai servizi multimediali in mobilità aumenterà in maniera esponenziale. Grazie ai Mobile Edge Server distribuiti in prossimità dell'utente mobile tali servizi diventeranno possibili abbattendo i limiti attuali di una tipica architettura centralizzata quali per esempio il ritardo introdotto nella trasmissione dei contenuti dagli attuali cloud dislocati nella core della rete che risulta essere incompatibile con i requisiti di molti servizi multimediali. Questo lavoro è una 'proof of concept' (PoC) su come l'utilizzo combinato di tecniche di machine learning in scenari MEC possano dar luogo ad un connubio di reciproco vantaggio. A conferma di ciò, è stato recentemente definito il concetto di 'Mobile Edge Learning' (MEL) nel quale MEC interagisce con intelligenza artificiale (AI), nel senso che modelli di apprendimento, parametri e dati sono distribuiti su diversi edge servers e un modello di AI è addestrato su tali dati distribuiti. Un modello di apprendimento distribuito, noto come '*Federated Learning*', nel quale un nodo funge da orchestratore che aggrega localmente i parametri derivati e li restituisce aggiornati ai server distribuiti sulla rete che fungono da 'addestratori'. (36)

In particolare ci siamo posti il problema di poter verificare in tempo reale lato client il rispetto della risoluzione video che, per esempio, era stata preventivamente concordata col content provider, mediante un opportuno accordo commerciale denominato *'Service Level Agreement'* (SLA), per la fruizione di contenuti multimediali ad alta definizione in mobilità. L'approccio proposto è di tipo *'protocol-agnostic'* basato cioè sul solo monitoraggio di opportune metriche di memoria di un dato processo lato client e con l'impiego di machine learning riusciamo a dedurre, al variare del tempo, il cambio di risoluzione video del contenuto in esecuzione nel lettore multimediale.

4.1 INTRODUZIONE

Con le reti cellulari di prossima generazione 5G, il ruolo del Mobile Edge Computing diventerà sempre più importante per potere usufruire in mobilità di servizi a bassissimo delay come il gaming online e realtà virtuale come pure quelle che richiedono intensi calcoli che un client sia per limitate capacità computazionali che per consumo di energia non potrebbero svolgere appieno. Tra questi servizi, uno davvero popolare è quello dello streaming in alta definizione. Secondo una recente indagine condotta da Cisco Visual Networking Index (37) si stima che entro il 2020, il 75% del traffico dati mobile a livello globale sarà generato dai servizi streaming. Dato che questi servizi diventeranno sempre più parte integrante della nostra vita sociale, anche le aspettative degli utenti sulla *'Quality of Expierence'* (QoE) crescerà.

Nonostante ciò, la fruizione di video streaming ad alta definizione su una rete mobile ha ancora diverse sfide da superare. Primo, far rispettare una concordata *'Quality of Service'* richiede una larghezza di banda stabile. Secondo, qualunque contenuto multimediale richiede una buona capacità di risorse computazionale per svolgere operazioni di codifica e decodifica sia nel lato trasmittente che ricevente. Terzo, valori di delay e jitter devono essere mantenuti al di sotto di precisi valori per poter assicurare la fluidità del servizio. Tutti questi problemi sono, ad oggi, stati affrontati dal punto di vista della rete. Il lavoro, che stiamo per descrivere, è, al meglio della nostra conoscenza, il primo che si concentra nel monitoraggio della qualità della risoluzione video dal punto di vista del client mobile in un ambiente di MEC. Immaginiamo uno scenario in cui un client abbia pagato un servizio premium per contenuti multimediali ad alta risoluzione secondo precisi SLA. In aggiunta, immaginiamo anche che i servizi saranno forniti in '*forma chiusa*', ovvero tali che uno sniffer di

protocollo non possa dedurre caratteristiche salienti su di esso. Dimostreremo che solo monitorando metriche di memoria, disponibili in un qualsiasi sistema operativo di tipo Unix, lato client sarà possibile riconoscere in tempo reale l'attuale risoluzione video del nostro servizio streaming, senza il bisogno di accedere ad informazioni di livello applicativo. Proponiamo, dunque, una soluzione basata su machine learning che consiste nella progettazione e sviluppo di un classificatore di risoluzione video capace di riconoscere la corretta risoluzione video soltanto monitorando tracciati di memoria. La Figura 26 mostra un possibile scenario nel quale la nostra soluzione potrebbe essere impiegata. Immaginiamo la possibilità che, per esempio, un client possa innescare la migrazione del servizio verso un altro mobile edge server a causa della violazione del SLA.

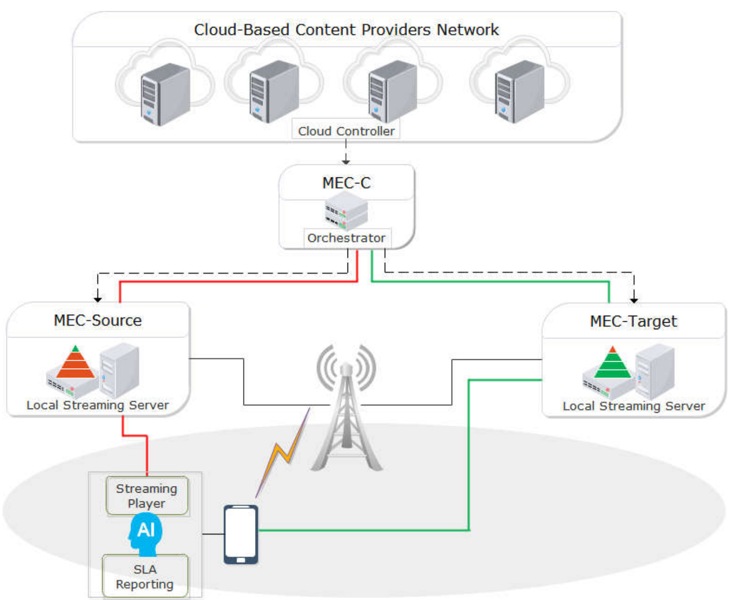

Figura 26 - Service Handover causa violazione SLA innescato dal client mobile

La Figura 27 mostra, invece, i messaggi di segnalazione tra client, MEC sorgente, MEC target e controller MEC relativamente ad una violazione dei termini di SLA che fanno scattare lato client la richiesta di un service handover.

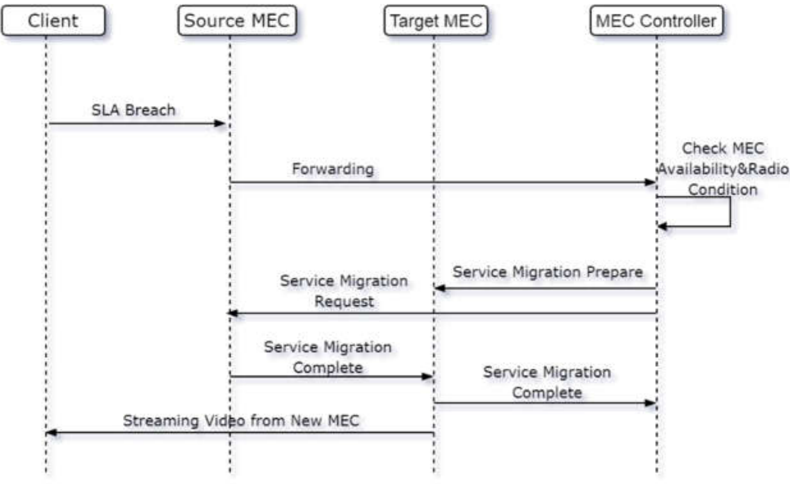

Figura 27 - Client-Side Service Handover

Riepilogando, grazie a questo lavoro, portiamo i seguenti contributi:

1. Estendiamo un tool per monitorare in tempo reale diverse metriche di memoria di un processo
2. Dimostriamo che il '*Working Set Size*' (WSS) non è fondamentale per dedurre la risoluzione video
3. Progettiamo, sviluppiamo, ottimizziamo e validiamo una rete neurale 'Multi Layer Percepetron (MLP)'

Infine questo lavoro di ricerca è stato presentato alla conferenza di Roma: "*The Fourth IEEE International Conference on Fog and Mobile Edge Computing (FMEC 2019)*"

4.2 Letteratura su Lavori Precedenti

Come già evidenziato precedentemente, vi sono molte soluzioni lato rete per migliorare la qualità dello streaming in ambiente MEC. In (38) gli autori propongono uno schema per adattare dinamicamente un selezionato bitrate basandosi sul monitoraggio in tempo reale della qualità video percepita. In (39) gli autori utilizzano sia Network Function Virtualisation (NFV) e MEC per trasmettere in modo efficiente contenuti multimediali in Ultra High Definition (UHD). In (40) gli autori sfruttano le capacità computazionali e di storage dei MEC server per migliorare le prestazioni dello streaming tramite un bitrate di tipo adattivo. Ossia, a seconda delle condizioni della rete e delle capacità del dispositivo mobile si fornisce il contenuto multimediale ad un determinato bitrate. In più dettaglio, hanno analizzato in modo congiunto il problema del caching e dell'elaborazione allo scopo di minimizzare il costo di backhaul per video streaming on-demand. Tutti MEC server nelle vicinanze del source MEC, possono venire in aiuto nel momento in cui esso è carico, fornendo sia il video richiesto via caching che svolgendo la transcodifica al desiderato bitrate. In (41) gli autori propongono un meccanismo che permette ad un SDN controller di fornire la corretta larghezza di banda e selezione del percorso di traffico assistendo gli utenti a selezionare un ottimale livello di qualità video. Concludendo, tutti questi lavori non sfruttano le nuove tecniche di machine-learning né tantomeno considerano tracciati video reali nei loro esperimenti.

4.3 Progettazione del Classificatore

La progettazione del nostro classificatore video ha riguardato due fasi principali: La prima, in cui presentiamo la metodologia per estrarre le metriche di memorie descritte nel paragrafo 1.5.2 e selezionare le 'features' più adatte per il nostro problema di classificazione. Infine andremo ad ottimizzare il classificatore progettato.

4.3.1 Estrazione Dati e Stima del Working Set Size

Per ottenere la nostra sorgente dati siamo partiti da un'interessante lavoro (42) che permette di stimare il '*Working Set Size*' (WSS) estendendolo per includere altre metriche di memoria che potrebbero essere discriminanti per il nostro classificatore. La stima del WSS è stata svolta utilizzando sia '*/proc/<PID>/clear_refs*' che '*proc/<PID>/smaps*'. In dettaglio, prima di effettuare una nuova misura, resettiamo i bit '*PG_Referenced*' e '*ACCESSED/YOUNG*' di tutte le pagine di memoria associate al processo in esame di modo che possiamo tracciare i loro aggiornamenti. Dopo

ogni fissato intervallo di misura , che abbiamo impostato pari a 10 ms, leggiamo la page table per monitorare quali indirizzi di memoria sono stati acceduti/scritti durante l'intervallo di misura andando ,dunque, a stimare la quantità di memoria utilizzata per dato processo. Bisogna tenere in considerazione il fatto che l'esecuzione di tali operazioni di reset e lettura della page table richiedono del tempo (nel caso ideale sono operazioni istantanee), dunque, l'intervallo di tempo totale per leggere il file è maggiore dell'intervallo di misura. Ciò implica che il valore fornito sul numero di pagine che sono state modificate potrebbe essere maggiore di quello corrispondente al nostro intervallo di misura. Nello specifico ciò che realmente succede dal punto di vista temporale è:

1. Inizio reset page flags delle pagine associate al processo
2. Trascorrono cicli CPU
3. Reset completato
4. Sleep per fissata durata
5. Inizio lettura page flags
6. Trascorrono cicli CPU
7. Lettura completata

Dunque, la durata è stimata misurando a partire dal punto centrale del passo 2 sino al punto centrale del punto 6. Per piccoli processi, essa corrisponderà probabilmente con l'intervallo di misura desiderato, mentre per processi estremamente grandi, che occupano più di 10 GB di spazio, potrebbero impiegare più di 500 ms di tempo di CPU e, dunque, una durata desiderata di 10 ms può realmente corrispondere a 100 ms

Esso fornisce una serie di metriche che sono: 'RSS', 'PSS','REF' ovvero 'WSS'. Abbiamo, dunque, esteso tale script aggiungendo le seguenti metriche:

- $USS = Private\ Dirty\ [MB] + Private\ Clean\ [MB]$
- $Private\ Dirty\ [MB]$
- $Shared\ Dirty\ [MB]$
- $Dirty\ Page = Private\ Dirty\ [MB] + Shared\ Dirty\ [MB]$

Per concludere, per fissato periodo di osservazione D lo script mostrerà ad ogni intervallo di misura $T_{sampling}$ una riga con sette colonne ciascuna indicante il valore della corrispondente metrica di memoria. Il codice sorgente è presente in APPENDICE B1.

4.3.2 Feature Analysis

Per la nostra analisi abbiamo considerato quattro classi di risoluzione video: '*Standard Definition*', '*HD Ready*', '*Full HD*' e '*UHD*' Queste classi corrispondono approssimativamente ad una richiesta di larghezza di banda di $3, 5, 10\ e\ 25\ Mbit/s$ .Anziché monitorare il loro throughput a livello applicativo, abbiamo analizzato la possibilità di identificare tali classi di risoluzione video per mezzo delle metriche di memorie corrispondenti al nostro video streaming. Per fare ciò, abbiamo costruito un dataset costituito dai files ognuno appartenente a differenti categorie e contenenti i tracciati delle metriche di memoria ottenuti durante la loro riproduzione. Successivamente, costruito il dataset, abbiamo analizzato gli scatter plots su un piano bidimensionale andando a considerare tutte le possibili combinazioni di due features su un totale di sette metriche. La Figura 28 mostra

soltanto tre esempi di come le features si distribuiscono tra loro rappresentando con simboli e colori differenti le diverse classi.

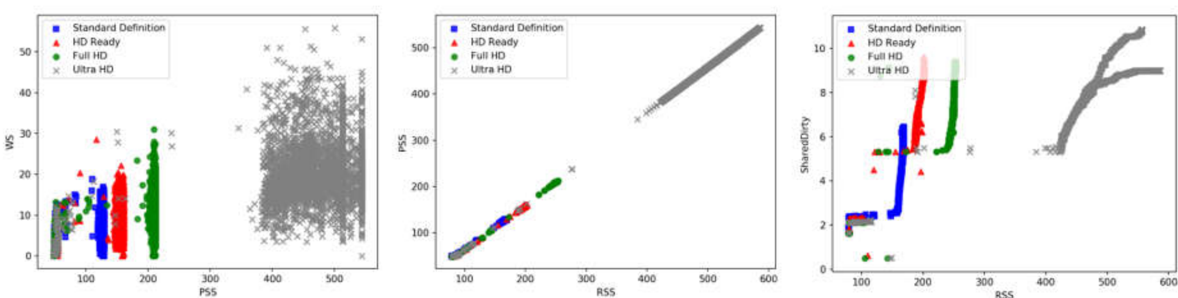

Figura 28 - Analisi Distribuzione Dati delle Features

Dall'analisi emerge che i punti spesso si sovrappongono tra loro facendo sì che le regioni di decisione di ciascuna classe non siano linearmente separabili. La successiva analisi è stata quella di valutare l'importanza di ogni feature, intesa come la capacità di essere discriminante ai fini della corretta classificazione della classe. Questa necessità di filtrare dal nostro dataset il numero di features nasce dall'osservazione che alcune delle metriche di memoria introdotte sono date dalla somma di altre. Esistono diverse tecniche di '*feature selection*'. Quella impiegata in questo lavoro è una tecnica d'insieme che si basa su '*random forest*' (43). Utilizzando tale tecnica possiamo misurare l'importanza delle features come la riduzione media delle impurità, calcolata da tutti gli alberi decisionali della foresta senza effettuare alcun supposizione sul fatto che i dati siano separabili linearmente oppure no. La Figura 29 mostra un interessante risultato ottenuto dalla 'feature selection' e cioè che il working set size (WSS) risulta essere il meno discriminante al fine di predire la corretta risoluzione video. Questo risultato lo si può giustificare andando a guardare l'alta varianza che ha il WSS paragonata a quella delle altre features. Inoltre, come già evidenziato prima, considerato il fatto che alcune metriche dipendono da altre, come ad esempio '*USS*' che dipende da '*Private Dirty*', abbiamo deciso di considerare il seguente sottoinsieme di features per la progettazione del nostro classificatore di risoluzione video. $[PSS - RSS - Private\ Dirty - Shared\ Dirty]$.

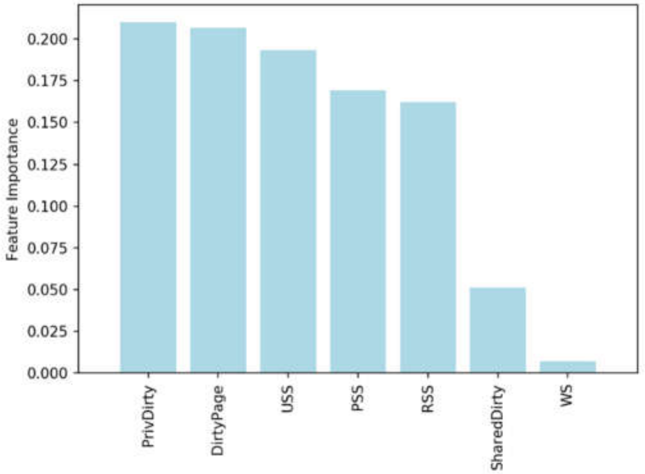

Figura 29 - Feature Selection

4.3.3 DISEGNO DI UNA RETE NEURALE MULTILAYER PERCEPTRON

Dato che i nostri dati non sono linearmente separabili abbiamo considerato per la nostra progettazione una rete neurale di tipo '*Multilayer Perceptron*' (MLP) (44). Un buon disegno di tale rete richiede di determinare un opportuno layout, inteso come numero di livelli nascosti e numero di neuroni per ciascun livello come pure i pesi associati ai link tra i neuroni. Avendo solo quattro features d'ingresso e identificato, dunque, un numero limitato di regioni di decisione pari al numero di classi e cioè quattro. Abbiamo, dunque, considerato una rete MLP costituita da due soli livelli nascosti[2]. Il primo avente tanti neuroni quante sono le classi e corrispondente all'uscita del nostro classificatore ed un secondo livello nascosto la cui dimensione in termini di neuroni è stata ottimizzata andando a valutare le prestazioni ottenute in termini di accuratezza con un diverso numero di neuroni. Per poter, dunque, determinare il miglior classificatore abbiamo costruito una *'pipeline'*, ossia una serie di step eseguiti in cascata ciascuno dei quali costituito una coppia di operazioni di 'fit' (adatta) e 'transform' (trasforma) il cui output diventa l'input dello step successivo sino all'ultimo step che sarà costituito da un'estimatore rappresentante il nostro modello predittivo. La Figura 30 mostra il funzionamento di una generica pipeline.

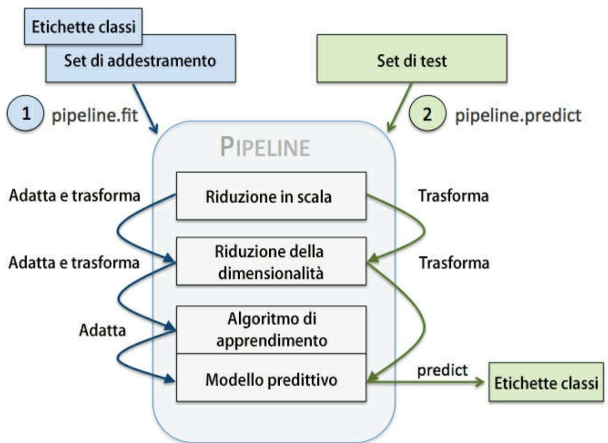

Figura 30 - Pipeline (Fonte: Machine Learning con Python)

Nel caso specifico, avendo già eseguito a monte la feature selection, la pipeline risulta essere costituita da due soli steps quello di standardizzazione delle features e quello costituito dal nostro algoritmo di apprendimento ossia estimatore. In più dettaglio, poiché un modello può soffrire di '*underfitting*' (elevato bias), se il modello è troppo semplice, o di '*overfitting*' dei dati di addestramento (elevata varianza) se il modello è troppo complesso. Per potere determinare un modello ottimo cioè che fosse un buon compromesso tra bias-varianza abbiamo utilizzato la tecnica di convalida incrociata K-fold ('*cross-validation*'), andando a valutare le prestazioni del modello, tramite un brute-force, in cui sono state considerate tutte le possibili combinazioni dei valori associati ad ogni iperparametro disposti in una griglia di ricerca ('*grid-search*'). La seguente Tabella 2 mostra la nostra iniziale griglia di ricerca dove la prima colonna contiene il nome dell'iperparametro, mentre la seconda mostra l'intervallo di valori scelto per la nostra grid-search. Il significato dei vari iperparametri è auto-esplicativo, il primo infatti è il numero di neuroni del livello nascosto, il secondo rappresenta la funzione d'attivazione, il terzo è lo specifico algoritmo

---

[2] In realtà la rete neurale consta in un solo livello nascosto, solo che in letteratura alcuni considerano anche il livello d'uscita. Ecco, spiegato del perché si parla di due livelli nascosti

utilizzato per l'ottimizzazione dei pesi, il quarto è il termine di 'regolarizzazione' ed infine l'ultimo rappresenta il tipo di learning nell'aggiornamento dei pesi. Alla fine di tale processo la migliore combinazione di iperparametri calcolato sul dataset di addestramento è mostrato nella Tabella 3

Il codice sorgente per progettare il nostro classificatore è in APPENDICE B2

Nome Iperparametro	**Intervallo di Valori**
Hidden Layer Sizes	$[(3,),(4,),(5,),(6,),(7,),(8,)]$
Activation Function	$['logistic','tanh','relu']$
Solver	$['lbfgs','sgd']$
Alpha	$0.5, 0.3, 1*e^{-1}, 1*e^{-2}, 1*e^{-3}, 1*e^{-4}, 1*e^{-5}$
Learning Rate	$['constant','adaptive']$

Tabella 2 - Grid Search

Nome Iperparametro	**Valore Ottimo**
Hidden Layer Sizes	7
Activation Function	$tanh$
Solver	$lbfgs$
Alpha	$1e^{-1}$
Learning Rate	$constant$

Tabella 3 - Valori Ottimi Iperparametri

4.4 Valutazione delle Prestazioni

Dopo aver introdotto il significato delle metriche e spiegata la metodologia impiegata. Andremo adesso a spiegare in dettaglio le due nostre sorgenti dati, il nostro ambiente sia di test che software ed infine andremo a discutere delle prestazioni del nostro classificatore in termini di score e matrice di confusione.

4.4.1 Data Sets

Abbiamo creato due data sets: Il primo, costituito da 8 files 2 per ogni classe , per un totale di 14667 campioni, che abbiamo chiamato '*Old Dataset*' ed un secondo costituito da 20 files (4 Standard Definition (SD) , 7 HD Ready, 4 full HD e 5 UHD) per un totale di 43869 campioni, che abbiamo chiamato '*New Dataset*'. Figura 31 mostra la distribuzione per classe di entrambi i datasets. Old Dataset ha praticamente un'equa distribuzione per classe, mentre New Dataset è circa 3 volte più grande del primo, ed è leggermente sbilanciato in numero di campioni HD Ready. Più in dettaglio Old Dataset è stato impiegato sia per l'individuazione del sottoinsieme ottimo di features che per il disegno del layout della rete MLP, mente New Dataset è stato utilizzato per ottimizzare la rete MLP e testare la sua capacità a generalizzare.

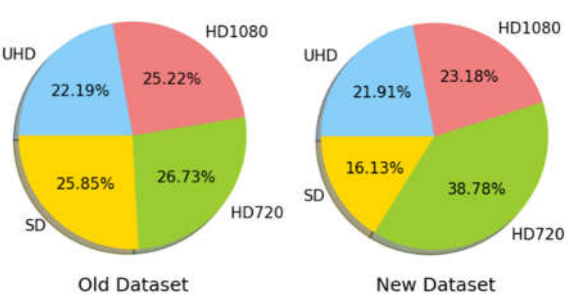

Figura 31 - Data Distribution

4.4.2 Ambiente di Test e Software

Per svolgere questo lavoro abbiamo realizzato due ambienti: uno di test ed un altro per sviluppare il nostro classificatore. Il primo è una virtual machine in ambiente Linux in cui girava lo script Perl per estrarre le metriche di memoria durante i test di riproduzione audio video così da formare poi i nostri datasets, mentre il secondo era un ambiente di sviluppo in Windows in cui era installato tutto il software necessario per sviluppare il classificatore. I dettagli di entrambi gli ambienti sono riassunti tramite le due seguenti tabelle: Tabella 4 e Tabella 5 .

Sistema Operativo	Xubuntu 64 bit
Kernel	4.15.0-36-generic
RAM	2 GB
Accelerazione Hardware	VT-x, KVM Para virtualizzazione

Scheda Video	64 MB
Scheda di Rete	Intel PRO/1000 MT Desktop (NAT)

Tabella 4 - Test Environment

Ambiente di Sviluppo	PyCharm Professional 2108.3
Linguaggi di Programmazione	Python 3, Perl
Librerie Software	Scikit-learn (45) e Mlxtend (46)
Lettore Multimediale	VLC 3.0.3

Tabella 5 - Software Environment

In tutti i test abbiamo impostato un tempo di campionamento $T_{Sampling} = 0.01\ s$, nessuna pausa tra una misura e la successiva $S = 0$ ed una durata dello streaming D maggiore della sua effettiva durata per tener conto del tempo impiegato nel selezionare, eseguire il file e chiudere l'applicazione.

4.4.3 RISULTATI NUMERICI

Una volta scelto il nostro sottoinsieme di features, abbiamo utilizzato un approccio empirico finalizzato a validare la nostra scelta come metriche di input. Per far ciò, partendo dalla nostra rete MLP ottima , costituita da $7\ e\ 4$ neuroni nei due livelli nascosti abbiamo di volta in volta considerato differenti sottoinsiemi di features e per ogni combinazione abbiamo valutato l'*accuratezza*, che ricordo essere : "*La Percentuale di classificazioni corrette*". Per il suo calcolo si ricorre all'utilizzo di:

- TP Numero di veri positivi (Classificati true erano true)
- TN Numero di veri negativi (Classificati false erano false)
- FP Numero di falsi positivi (Classificati true anche se false)
- FN Numero di falsi negativi (Classificati false anche se true)

$$\frac{TP + TN}{TP + TN + FP + FN}$$

Figura 32 mostra il valore di accuratezza raggiunta per le più significative combinazioni. Come si evince dal grafico il migliore valore di accuratezza è raggiunto dall'ultima combinazione, bar gialla, che rispetto a quella da noi scelta presenta però una feature in più (WSS) a fronte di un minimo aumento di accuratezza pari al 0.2% tale da non giustificare un aumento di complessità del nostro classificatore.

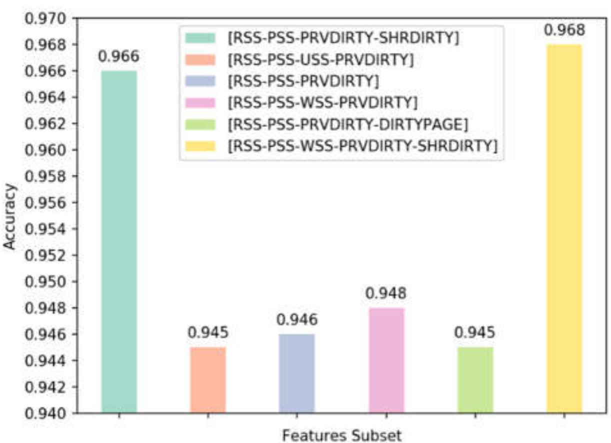

Figura 32 - Validazione Features

Dopo aver, dunque, validato il nostro sottoinsieme di features, il secondo step è stato quello di valutare la robustezza della nostra rete MLP ovvero che abbia raggiunto una capacità a generalizzare esente da possibili overfitting. I test sono stati svolti considerando diverse strutture di reti neurali MLP, ed in particolare, abbiamo considerato reti sia con numero di differenti di livelli, da uno a tre, che con diverso numero di neuroni per ciascun livello. e di volta in volta quantificato l'accuratezza applicando sempre la cross-validation e come sorgente dati il New Dataset. Il motivo che ci ha indotti a validare il nostro layout di rete è stato dovuto al fatto che poiché il dataset con cui avevamo determinato la rete ottima era stato costruito unendo diversi file dati in uno soltanto, tramite un successivo split random del dataset ,per determinare la parte relativa al training e quella di testing ,implica con quasi certezza deterministica che almeno una riga dati di ogni file sia presente nel nostro training set e quindi il classificatore poteva in qualche modo imparare il pattern a memoria piuttosto che apprendere. Ecco, quindi, che abbiamo costruito un secondo dataset che il classificatore non aveva mai visto prima e valutato l'accuratezza per differenti strutture di reti MLP. In dettaglio, Figura 33 mostra il valore di accuratezza ottenuta considerando soltanto i test data dell' Old Dataset (barre verdi) , quella ottenuta considerando l'intero New Dataset (barre rosse) e il valore medio del cross-validated score, cioè l'accuratezza media calcolata sui K validation sets dell' Old Dataset.(barre). Infine la curva misura il degrado dell'accuratezza, ossia la caduta di accuratezza rispetto all' Old Dataset espressa in percentuale. La cui formula è la seguente:

$$\left[\frac{Accuracy_{new_{dataset}} - Accuracy_{old_{dataset}}}{Accuracy_{old_{dataset}}}\right] * 100$$

(1) - Accuracy Drop

Figura 33 evidenzia anche come un singolo livello nascosto con 7 neuroni fornisca i migliori risultati, con la minima caduta d'accuratezza pari a circa -1.5%. Notiamo anche che per il nostro problema di classificazione, anche le reti con multi livelli nascosti (Deep Neural Network) forniscono ottimi risultati quando testati con campioni random dell' Old Dataset , ma poi si dimostrano essere molto sensibili a potenziali problemi di overfitting quando testate con nuovi dati e cioè col New Dataset. Infine, abbiamo anche notato che la topologia di rete MLP selezionata

rispetta una serie di comuni regole empiriche (47) (48) (49) che sostanzialmente dicono che il numero di neuroni dovrebbe essere un numero pari alla somma di $^{2}/_{3}$ del numero di features in ingresso più il numero delle variabili d'uscita. I risultati ottenuti, confermano anche il *'Teorema di Approssimazione Universale'* (50) ovvero che un singolo livello nascosto è sufficiente per approssimare un'arbitraria funzione continua di n variabili reali. Questi risultati, dunque, suggeriscono che il miglior design per un classificatore video che si basa su metriche di memoria è quello con un singolo livello nascosto con un numero di neuroni da tenere il più basso possibile per evitare possibili problemi di overfitting.

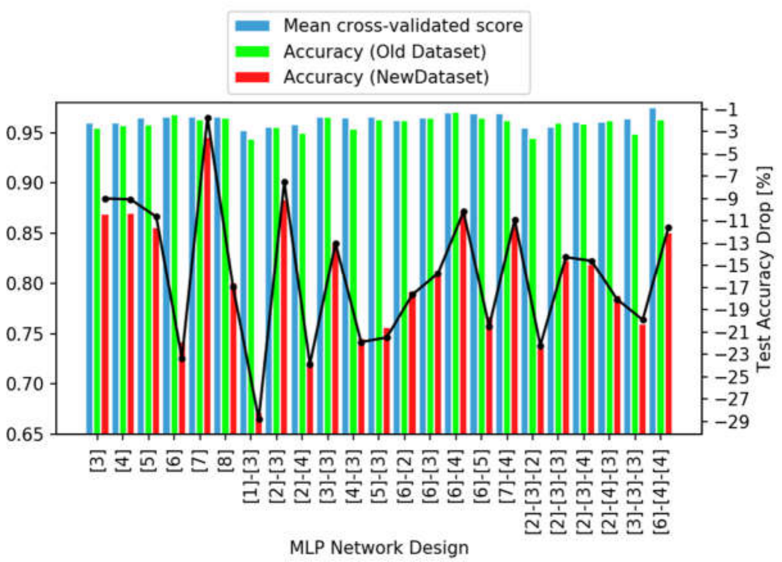

Figura 33 - Design Ottimo Rete MLP

Per concludere abbiamo calcolato la matrice di confusione, mostrata in Tabella 6, del nostro classificatore ottimo. I risultati mostrano che il classificatore ha raggiunto, in generale, un eccellente capacità di predire la risoluzione video con un valore di accuratezza che oscilla tra il 95% e il 96% su dati mai visti prima. Gli errori di classificazione variano da un minimo del 0.08% ad un massimo del 5.3% .In particolare, i più alti errori di classificazione sono quelli dei campioni SD predetti come HD720 pari a quasi il 4.5% e campioni appartenenti alla classe UHD predetti invece come HD1080 pari circa a 5.3% .

Classe Reale	**Classe Predetta**	SD	HD720	HD1080	UHD
SD		1091	52	0	0
HD720		19	1149	0	0
HD1080		7	8	1129	1
UHD		10	2	50	883

Tabella 6 - Matrice di Confusione

4.5 Riepilogo

Riepilogando, in questo lavoro, abbiamo progettato e sviluppato un classificatore di risoluzione video che potrebbe essere installato come un'applicazione negli smartphones per poter monitorare SLA del servizio di video streaming stabilito col content provider. L'idea è quella di innescare in modo proattivo un service handover qualora la qualità della risoluzione video dovesse scendere ad un livello inferiore da quello pattuito. Il nostro disegno '*data-driven*' ha dimostrato che è possibile classificare correttamente un video solo affidandosi a specifiche metriche di memoria. Abbiamo, inoltre provato che, se pur di estrema importanza per ogni Sistema Operativo, il *WSS* non è risultato discriminante per discernere la corretta risoluzione. Abbiamo, inoltre, presentato una metodologia per collezionare i tracciati di memoria e costruire il classificatore, discutendo anche le nostre scelte progettuali e la sua robustezza. Per quest'ultimo aspetto, abbiamo impiegato la *cross-validation* dove avevamo preliminarmente diviso i nostri files video in due gruppi (uno per il training e l'altro per valutare la sua robustezza nel predire la corretta della risoluzione video). Un futuro lavoro potrà essere quello di generalizzare il modello considerando differenti kernel e lettori multimediali ed investigare in dettaglio sul perché, nonostante l'accuratezza complessiva sia alta, ci siano errori tra il 4% e il 5% nel classificare i campioni delle classi SD e UHD rispettivamente.

Infine, un'ultima considerazione, è la possibilità di discernere le caratteristiche di un servizio/applicazione, anche se il sorgente non è aperto, senza alcuna necessità di analizzare nessuno messaggio di protocollo. Una tale situazione potrebbe portare ad un potenziale problema di sicurezza che necessiterà essere analizzato in futuro.

5 OTTIMIZZAZIONE PRE-COPY: ARCHITETTURA RATAMCA

Con l'emergere di nuovi servizi in mobilità, la continuità del servizio giocherà sempre più un ruolo chiave. Mentre tecnologie come Mobile Edge Computing (MEC) permettono l'esecuzione di applicazioni prima non possibili fornendo le necessarie risorse hardware e software in prossimità del client mobile, resta il problema, a causa sia della mobilità che delle risorse variabili nel tempo, di migrare il servizio il più velocemente e trasparente possibile. Dopo aver analizzato le potenzialità della tecnica di pre-copy per applicazioni RAM-intensive, estendiamo il lavoro precedente cercando di ottimizzare i tempi di migrazioni. A tale scopo, disegniamo e modelliamo una nuova architettura MEC che sfrutta la riconfigurazione dinamica dei percorsi di rete

5.1 INTRODUZIONE

La nuova ed emergente rete cellulare 5G insieme alle tecnologie cloud dislocate in prossimità dell'utente mobile, come per esempio il MEC, permetterà l'esecuzione di nuove applicazioni quali gaming online, realtà aumentata e high performance application (HPC) che hanno stringenti requisiti in termini di jitter, delay e calcolo. È abbastanza chiaro che la migrazione del servizio giocherà un ruolo determinante in tali circostanze. In generale la migrazione di un servizio coinvolge una o più virtual machines (VM) o containers. Abbiamo già parlato di quali siano i più noti ed importanti algoritmi di live migration che ricordiamo essere: pre-copy, post-copy e hybrid-copy. In questi ultimi anni, pre-copy ha destato particolare attenzione grazie alla sua intrinseca robustezza. Inoltre, per alcune particolari applicazioni la cui frequenza di sporcamento delle pagine di memoria è elevata, conosciute come applicazioni RAM-Intensive, l'ottimizzazione del pre-copy diventa necessaria e fondamentale. I lavori di ricerca sul pre-copy sono tutti incentrati nell'ottimizzare le particolari caratteristiche del processo coinvolto durante la migrazione. Alcuni esempi sono: investigare sulla similarità delle pagine di memoria, data deduplication, funzioni di checkpoint e restore, compressione dati e così via. La novità di questo lavoro consiste nel presentare un'architettura MEC che in stretta collaborazione con una rete di trasporto SDN migliori i tempi di migrazione del pre-copy. Vedremo che questa architettura tramite riconfigurazione dinamica dei percorsi di rete sia durante la fase iterativa che in quella finale di stop-and-copy, ridurrà i tempi di migrazione con conseguente miglioramento della stessa migrazione. Con questo lavoro dimostriamo che questa architettura, a seconda delle condizioni di traffico tra sorgente e destinazione può ottenere miglioramenti perfino superiori al 50% rispetto ad un instradamento statico. In particolare i contributi apportati sono:

1. Introduzione di una nuova architettura MEC e SDN
2. Modellizzazione e Simulazione di questa architettura
3. Analisi delle prestazioni

5.2 LETTERATURA SU LAVORI PRECEDENTI

Questo paragrafo sintetizza e confronta diverse soluzioni architetturali MEC proposte in letteratura. Inoltre riassume lo stato dell'arte delle soluzioni di ottimizzazione del pre-copy.

5.2.1 ARCHITETTURE MEC

Una prima architettura MEC, introdotta nel 2012 nel progetto European Tropic Project (51), fu la '*Small Cell Cloud*' (SCC). L'idea principale dietro questa architettura fu quella di potenziare dal punto di vista computazionale e di storage le small cells come microcells, picocells e femtocells. A tal fine fu introdotta una nuova entità denominata '*small cell manager*' (SCM) per gestire e coordinare le small cells installate. Principale funzione del SCM è gestire le risorse di storage e quelle computazionali. A seconda del tipo di deployment l'architettura può essere centralizzata o distribuita. Altra architettura MEC è *'mobile micro clouds'* (MMC) (52). Principale differenza con la precedente architettura è che questi MMC sono interconnessi tra loro e sono in rapporto 1 : 1 con le stazioni radio base. Questo consente una migrazione del servizio più fluida garantendo anche la continuità del servizio. Una terza architettura è la *'Fast Moving Personal Cloud'* (MobiScud) (53) che porta i servizi cloud per mezzo delle tecnologie SDN e NVF. Le risorse cloud sono piazzate nella rete di trasporto SDN e controllate da un'entita di nome MobiScud control (MC), che s'interfaccia sia con la rete mobile che con gli switch SDN e il cloud dell'operatore. Questa architettura presenta una maggiore latenza poiché le risorse cloud non sono direttamente collegate alle stazioni radio base. Un'altra proposta architetturale è *'Follow me Cloud'* (FMC) (54). L'idea che sta dietro questa soluzione è che le risorse cloud vadano seguendo gli spostamenti dei client mobili. Questa architettura introduce due nuove entità: un datacenter gateway (DC/GW) il cui compito è mappare i datacenter ai vari service/packet gateway (S/P-GWs), e un controller (FMCC) che può essere installato o in modo centralizzato o gerarchico con un global FMCC (G-FMCC) e una serie di controller locali (L-FMCC) per migliorare la scalabilità. Infine, l'architettura, denominata '*CONCERT*' proposta in (55) , sfrutta sia SDN che NVF per gestire le risorse di storage, computazionali e di comunicazione.

5.2.2 STATO DELL'ARTE: OTTIMIZZAZIONE PRE-COPY

L'algoritmo di pre-copy deve far fronte ad alcuni problemi quali: convergenza di migrazione e un grande network overhead. Il primo problema si presenta quando la velocità di sporcamento delle pagine di memoria (memory dirtying rate) è maggiore della larghezza di banda disponibile, mentre il secondo è causato dalla fase iterativa. Nonostante ciò, pre-copy è largamene impiegato nelle più note e diffuse piattaforme di virtualizzazione, come per esempio XEN (32), KVM (12) e VMware (56) grazie alla sua intrinseca robustezza. Molti lavori mirano ad ottimizzare il pre-copy mediante l'utilizzo di differenti tecnologie. Hacking e Hudzia (57) fanno uso della compressione delta per ridurre la durata della migrazione di virtual machines che eseguono applicazioni che fanno uso di grandi quantità di RAM e hanno un elevato memory dirtying rate. Svärd ed altri, anziché far uso di '*Adapative Replacement Cache*' (ARC) (58) impiegano uno schema di cache associativa bidirezionale al fine di memorizzare le pagine referenziate. Riteau ed altri (59) disegnano un sistema di migrazione basato sulla deduplicazione dei dati per migliorare le prestazioni della migrazione della memoria su una WAN. Tale sistema si basa un indirizzamento distribuito del contenuto per evitare di trasmettere pagine duplicate tra la virtual machine migrata e quella a

destinazione. Wood ed altri (60) impiegano sia la compressione dati delta che la deduplicazione sempre su una WAN. Huang ed altri (61) sfrutta una funzionalità presente in InfiniBand, Remote Direct Memort Access (RDMA) (62) per ottimizzare la service migration. Liu ed altri (63) impiegano un nuovo sistema di migrazione basato su checkpoint e recovery e trace/replay (CR/TR). Svard ed altri (64) sfruttano il concetto di riordinamento pagina assegnando a ciascuna di loro un peso in funzione della loro frequenza di aggiornamento. Altri lavori provano a risolvere il problema della convergenza della migrazione, come Jin ed altri (65) i quali propongono di abbassare il downtime ottimizzando la frequenza delle VCPU perché loro scoprono che la frequenza di sporcamento delle pagine di memoria ha una relazione lineare con la velocità di esecuzione della VM. Infine, Atif e Strazdins provano a migliorare la migrazione di applicazioni HPC che sono solite avere un elevato dirtying rate. Essi affermano che copiare iterativamente la memoria per tali applicazioni è solo uno spreco di tempo e cicli CPU. Quindi, propongono di fare un pre-copy con soli due rounds. Il primo copia tutte le pagine di memoria ed il secondo va direttamente nella fase finale di stop-and-copy.

5.3 Soluzione Proposta

L'idea di proporre l'architettura, denominata Resource and Traffic Aware Mobile Cloud Architecture (‘*RaTAMCA*’), data dalla cooperazione di MEC e SDN nasce dopo aver letto (66). Gli autori calcolano sperimentalmente il tempo di riconfigurazione di una rete SDN che è definito come: *'Il tempo che va dall'invio del comando di riconfigurazione fino a quando l'effetto è visibile nel data plane'*. Dai loro esperimenti si ha che in caso di un'architettura con soli OpenVswitch (ovs) (switch virtuale) il tempo di riconfigurazione è indipendente dalla strategia di popolamento delle regole di flusso (inoltro) presenti nella tabella di ciascun dispositivo d’inoltro e tali tempi si attestano tra $1.25\ ms\ e\ 1.31\ ms$. Questi valori risultano, sulla base dei nostri precedenti risultati sull'analisi dei tempi, mediamente inferiori ai T_i rappresentanti i tempi di trasferimento delle singole iterazioni durante la fase di warm-up e quindi potrebbe avere senso definire un'architettura in cui i dispositivi di inoltro sono tutti virtuali allo scopo di ridurre i tempi di migrazione dell'algoritmo di pre-copy grazie all’ausilio di una rete SDN il cui controllo è logicamente centralizzato. Andremo, dunque, a modellizzare e simulare un controller SDN sotto vari scenari di test. Questo lavoro estende il lavoro svolto nella valutazione prestazionali di live migration.

5.3.1 Architettura di Rete

L’architettura proposta sfrutta la riconfigurazione dinamica dei percorsi offerta da una rete di trasporto SDN dell’architettura MobiScud (53). La nostra architettura ‘*RaTAMCA*’ mostrata in Figura 34 ha una topologia centralizzata costituita da una rete di server MEC co-locati con le base station in rapporto $1:1$ e connessi con i dispositivi di inoltro (FD) (router o switch) verso il controller SDN-C. In particolare si sta ipotizzando una migrazione dati da ‘source MEC’ a ‘target MEC’. SDN-C è integrato all’interno del MEC orchestrator e riceve i messaggi di controllo di livello 3 (L3) (colore rosso) da tutti i dispositivi FD. Sulla base di tali messaggi ricevuti, come per esempio la larghezza di banda disponibile su ogni FD nei collegamenti diretti con i loro peers, SDN-C può, in ogni momento, riconfigurare il percorso dati più veloce (colore nero) tra server MEC sorgente e destinazione indipendentemente dalla reale distanza fisica tra essi. Ossia, il percorso più veloce scelto dal controller può non corrispondere a quello fisicamente più breve. Ricordiamo che un’applicazione RAM-intensive è tale per cui la quantità di memoria principale coinvolta durante la sua esecuzione è più grande della quantità di spazio occupato dalla sua stessa installazione (memoria secondaria). In uno scenario di ‘*Live Migration*’ la dimensione dello stato della RAM sarà molto più grande di quello dei registri e della CPU. Dunque, la fase di warm-up e

non solo, dell’algoritmo di pre-copy necessita un’ottimizzazione al fine di ridurre il tempo complessivo di migrazione il più possibile permettendo una migrazione fluida e trasparente del servizio.

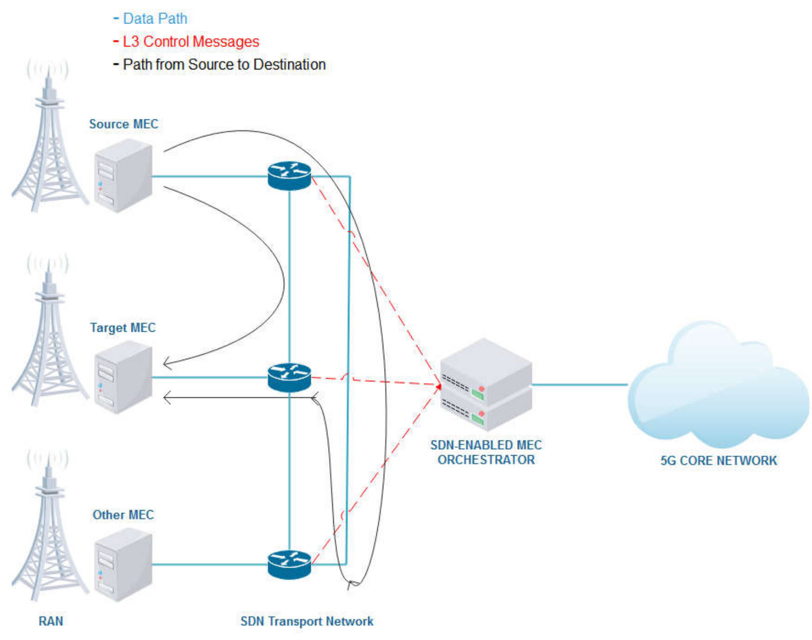

Figura 34 - RaTAMCA Architecture

Figura 35 mostra lo scenario di trasferimento delle pagine di memoria durante un’iterazione T_i della fase di warm-up nell’ipotesi di migrazione del servizio da un MEC sorgente (MEC-S) a quello destinazione (MEC-T). Per semplicità d’implementazione, abbiamo considerato una rete di trasporto con soli 3 FDs, così che la rete overlay totalmente connessa logicamente avesse soltanto 3 $\left(N\frac{N-1}{2}\right)$ collegamenti. Tale ipotesi, ai fini dei risultati non è limitante. In dettaglio, supponiamo che ogni MEC server sia connesso direttamente ad un FD ed essedo la topologia della rete overlay un triangolo, abbiamo soltanto due percorsi per data sorgente verso la destinazione. In particolare, siano MEC-S, MEC-T e MEC-X rispettivamente connessi a FD1, FD2, FD3. Ad ogni iterazione sulla base delle informazioni statistiche ricevute dai dispositivi d’inoltro FD (L3 stats), SDN-C potrà determinare tra tutte le possibili destinazioni tra MEC-S e MEC-T quella più veloce, ovvero il percorso col valore di T_i più basso o meglio quello che garantisce sulla base delle attuali condizioni di rete la maggiore larghezza di banda disponibile. Selezionato il percorso migliore SDN-C invierà le nuove regole di flusso ad ogni dispositivo FD coinvolto (messaggio di push new route) così che il possano acquisire la nuova configurazione d’instradamento e il MEC-S (data on new route) potrà inviare le pagine di memoria lungo il nuovo percorso più veloce. Il tempo di warm-up $\left(T_{warm-up}\right)$ è dato da (2)

$$T_{warm-up} = \sum_{i=1}^{n} T_{rsync}\,(i)$$

(2) Tempo di Warm-up

Dove ricordiamo che $T_{rsync}(i)$ rappresenta il tempo di sincronizzazione richiesto per trasmettere il numero di pagine che risultano '*dirty*' al termine della i-esima iterazione. Quindi, a seconda delle condizioni di traffico o meglio degli scenari di traffico prevediamo un netto miglioramento dei tempi di migrazione rispetto ad un instradamento statico tra sorgente e destinazione. Inoltre, poiché il tempo di downtime, secondo il lavoro precedente, dipende soltanto dal numero di pagine di memoria rimaste da trasferire diviso la larghezza di banda disponibile nel link di collegamento. Appare evidente che anche il tempo di downtime potrà beneficiare dall'instradamento dinamico delle pagine di memoria.

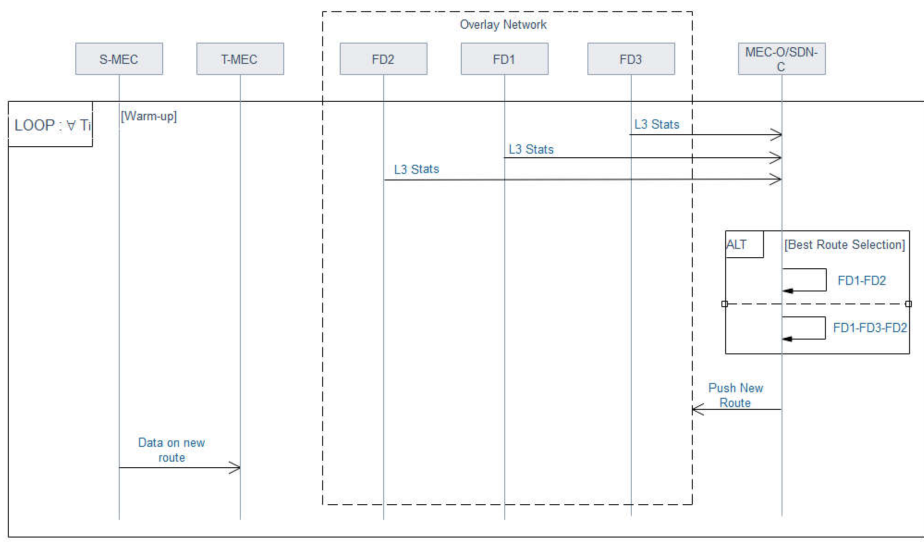

Figura 35 - Diagramma di segnalazione - Selezione del percorso più veloce

5.3.2 Modello di Simulazione

Come anticipato questo modello che andremo adesso a descrivere estende il precedente nel quale simulavamo l'algoritmo di pre-copy per applicazioni RAM-intensive. Come descritto prima, senza perdere di generalità, consideriamo tre FDs costituenti una rete overlay nella quale, per data destinazione, vi sono soltanto due possibili percorsi. Il primo percorso consta di un solo '*hop*', mentre il secondo ne ha due. Basandoci sul lavoro (67) che prova che il traffico di rete assume , tra diverse distribuzioni statistiche, quella di una distribuzione GEV, introduciamo un modello di larghezza di banda tempo variante applicato ad entrambi i percorsi. Definiamo la larghezza di banda disponibile $W_{AVBL}(x)$ come:

$$W_{AVBL}(x) = \begin{cases} W_{max} & x \le 0 \\ 0 & x \ge W_{max} \\ W_{max} - x & 0 < x < W_{max} \end{cases}$$

(3) - Modello di Traffico

Dove x è una funzione Matlab, '***gevrnd()***' (68), che ritorna un'array di numeri random scelti da una distribuzione GEV in cui fissiamo i parametri k, μ, σ i cui valori determineranno un preciso modello di traffico di rete. In dettaglio, il valore di μ simula la quantità di traffico (Alta, Media, Bassa) sul

percorso considerato , mentre W_{max} è la massima larghezza di banda impostata in un generico hop di un percorso. In aggiunta, simuliamo un SDN-C definito come:

$$T_{i_min} = \begin{cases} T_{i_p1} & T_{i_p1} \leq T_{i_p2} \\ T_{i_p2} & T_{i_p1} > T_{i_p2} \end{cases}$$

(4)- SDN Controller

In (4), T_{i_p1} (un solo hop di distanza dal server MEC destinazione) e T_{i_p2} (due hops di distanza dalla destinazione) rappresentano il tempo richiesto per trasferire le pagine di memoria che si sono sporcate al termine della i-esima iterazione lungo il percorso P_1 e P_2 rispettivamente. La funzione, dunque, di SDN-C è garantire prima di ogni iterazione della fase di warm-up, sulla base delle informazioni della corrente $W_{AVBL}(x)$ e le rimanenti pagine di memoria ancora da trasferire, il miglior T_i indirizzando l'algoritmo di pre-copy ad inviarle verso la destinazione col T_{i_min}.

5.4 Valutazione delle Prestazioni

Dopo aver spiegato la soluzione che intendiamo applicare, introduciamo tre modelli di traffico, descriviamo gli scenari considerati ed infine analizziamo i risultati numerici. La parte centrale del codice può essere visionata in APPENDICE C

5.4.1 Modelli di Traffico

Nel paragrafo 1.6 abbiamo introdotto alcuni concetti base sulla distribuzione GEV. Adesso, descriviamo e giustifichiamo la nostra scelta per i seguenti tre modelli di traffico '*Low - Medium - High*'. Tabella 7 e Tabella 8 mostrano i valori scelti per i parametri K, σ, μ rispettivamente nel caso in cui la capacità del link sia 1000 o 2000 $Mbit/s$.

Modello di Traffico	***K***	***σ***	***μ***
Low	0.2	2.4	31.5
Medium	0.2	2.95	52.3
High	0.2	2.7	85.5

Tabella 7 - Modelli di Traffico - Link = 1000 Mbit/s

Modello di Traffico	***K***	***σ***	***μ***
Low	0.2	2.4	63
Medium	0.2	2.95	104.6
High	0.2	2.7	171

Tabella 8 - Modelli di Traffico - Link = 2000 Mbit/s

Per ogni modello di traffico, abbiamo sempre scelto un valore positivo pari a 0.2 per K così che abbiamo una distribuzione con supporto infinito a destra (heavy-tailed a destra) , σ varia tra 2.4 e 2.95 così da limitare il più possibile la deviazione dal valore μ i cui valori variano tra 31.5 e 85.5 simulanti rispettivamente traffico di rete da 'Low sino ad High'. In dettaglio, un valore di μ pari a 31.5 equivale ad un traffico di rete tra il 25% e 35% della massima capacità del link, un valore pari a 52.5 indica un traffico di rete tra il 40% e 60%, mentre un valore pari a 85.5 è

equivalente ad un traffico di rete tra il 70% e 90%. Tabella 8 mostra gli stessi valori eccetto per il parametro μ i cui valori sono il doppio rispetto al caso precedente per tener conto del fatto che la capacità del link è il doppio (2000 $Mbit/s$).

5.4.2 Scenari di Test

Come descritto prima, la rete overlay ha due possibili percorsi per data destinazione o solo un hop (P1) o due hops (P2). Per ognuno di loro, abbiamo applicato il modello tempo variante della larghezza di banda (3) secondo uno dei tre modelli di traffico (Low, Medium, High) presentati in Tabella 7 e Tabella 8. Tabella 9 sintetizza, invece, i quattro scenari in cui: la colonna '*RAM*' indica un'applicazione RAM-intensive da 32 o 64 $[MB]$, '$W_{\max(P1)}$' indica la massima larghezza di banda lungo $P1$ e $W_{max(P2_L1)}$ e $W_{\max(P2_L2)}$ si riferiscono al percorso $P2$.

Scenario	*RAM* $[Mbit/s]$	$W_{\max(P1)}[Mbit/s]$	$W_{\max(P2_{L1})}[Mbit/s]$	$W_{\max(P2_{L2})}[Mbit/s]$	Num. Test
1	64	1000	1000	1000	5
2	64	1000	2000	1000	6
3	32	1000	1000	1000	6
4	32	1000	2000	1000	6

Tabella 9 - Scenari - Casi di Test

A secondo del modello di traffico applicato ad ogni percorso, abbiamo potuto simulare differenti casi di test. Abbiamo soltanto considerato pochi casi di test tra tutte le possibili combinazioni come indicato nella colonna '*Num. Test*'. Il nostro simulatore di pre-copy in congiunzione con SDN-C prende come ingresso la dimensione della RAM dell'applicazione, la massima larghezza di banda per ogni collegamento e il suo corrispondente modello di traffico. In uscita fornisce i tempi di 'warm-up', 'downtime' e 'total migration' in funzione della frequenza di sporcamento medio R_{avg}, che varia da 0 al massimo numero di pagine di memoria pari a $\left[{}^{RAM}/_{page_size}\right]$.
Fondamentalmente, il simulatore, per dato input, restituisce in un'uscita una riga per ciascuna variabile (warm-up, downtime e total migration) in funzione dei valori di R_{avg}. Per ciascun scenario, abbiamo ripetuto il test 3 volte, a parità di condizioni; il primo simulante un instradamento statico verso $P1$, il secondo verso $P2$ e poi usando SDN-C che, ad ogni iterazione, sceglieva dinamicamente dove inviare le pagine di memoria o su $P1$ o $P2$ o PX. Abbiamo ripetuto il medesimo scenario 100 volte, ottenendo quindi una matrice per ciascuna variabile d'uscita e per ciascun dei 3 casi prima descritti e calcolato infine sia la media che la mediana delle matrici. Abbiamo deciso di considerare soltanto i valori mediani così da rimuovere eventuali '*outlier*' che la distribuzione GEV poteva generare al di fuori del selezionato modello di traffico. Infine, per poter confrontare i risultati tra routing statico e dinamico, abbiamo costruito dei grafici dove in ordinata abbiamo messo la variazione percentuale definita come (5) in cui Y può assumere valore 1 o 2 ed in ascissa R_{avg}.

$$\left[\frac{T_{py} - T_{px}}{T_{py}}\right] \times 100$$

(5) - Variazione Percentuale

5.4.3 RISULTATI NUMERICI

La nostra analisi è stata svolta valutando gli scostamenti percentuali secondo la (5). Per ciascun dei quattro scenari abbiamo considerato solo i 4 più interessanti casi di test da investigare. Per dato scenario, da un test al successivo, diminuiamo la W_{AVBL} ad uno dei due possibili percorsi $P1$ o $P2$ e valutiamo eventuali benefici dell'instradamento dinamico. Le nostre considerazioni sono sul '*total migration time*', che è dato dalla somma di *'warm-up*' e '*downtime*'. Simili conclusioni possono essere ricavate per entrambe le variabili. Cominciamo, quindi, ad analizzare lo scenario 1. Figura 36(a) mostra il caso in cui $P1$ ha W_{AVBL} media, mentre in $P2$ è alta su entrambi gli hops. I risultati evidenziano che l'instradamento delle pagine di memoria avviene principalmente lungo il percorso $P1$, anche se la W_{AVBL}è maggiore lungo $P2$ che su $P1$. Infatti il confronto tra il percorso dinamico

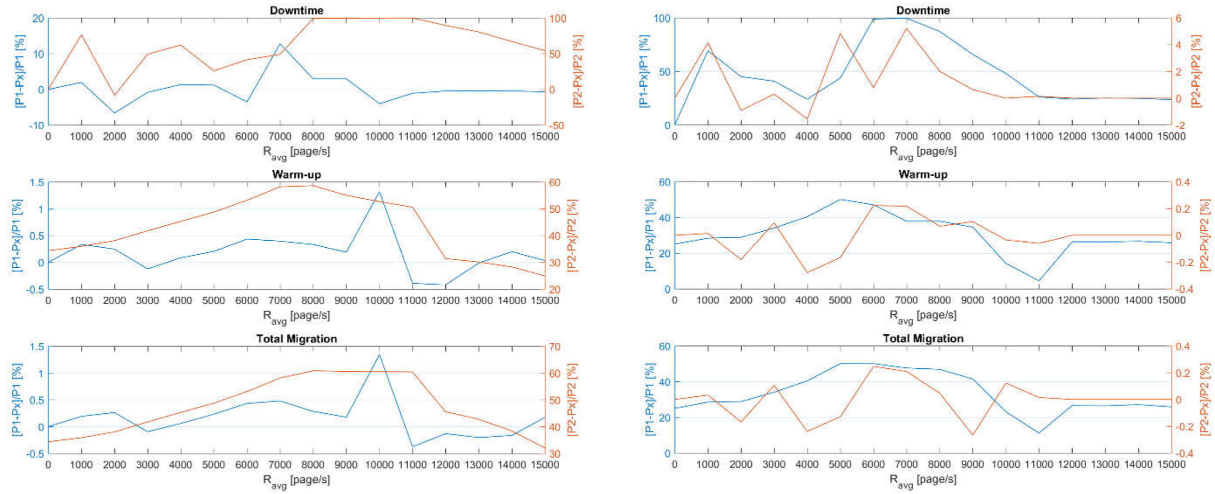

Figura 36 - (a) P1(M)-P2(H-H) - (b) P1(L)-P2(H-H) - Pre-copy Performance - Scenario 1

PX e $P1$mostra un miglioramento del '*Total Migration Time* 'praticamente nullo e cioè oscillante tra $0.2\% - 1.4\%$, mentre è tra $35\% - 60\%$ rispetto ad un istradamento lungo $P2$. In Figura 36(b) il pattern di W_{AVBL} è: 'Low-High-High'. A causa del peggioramento di W_{AVBL} lungo $P1$, le pagine di memoria sono instradate lungo $P2$. Il miglioramento del '*Total Migration Time*' rispetto a $P1$ oscilla tra $25\% - 55\%$. Figura 37(a) mostra un test case il cui W_{AVBL} pattern è: 'Low-High-Medium'. Il risultato ottenuto è simile al precedente, ossia un generale miglioramento del '*Total Migration Time*' tra il $15\% - 31\%$ rispetto a $P1$. Infine Figura 37(b) il cui W_{AVBL} è: 'Low-Medium-Medium', possiamo apprezzare i benefici dell'instradamento dinamico lungo entrambi i percorsi. Infatti, essendo W_{AVBL} bassa su entrambi i percorsi a causa dell'elevato traffico di rete, SDN-C sceglie, per ogni R_{avg}, tra i due possibili percorsi quello più veloce. In dettagli, otteniamo un miglioramento del '*Total Migration Time*' tra il $4.5\% - 14.8\%$ per $P1$ e tra $4.3\% - 4.9\%$ lungo $P2$.

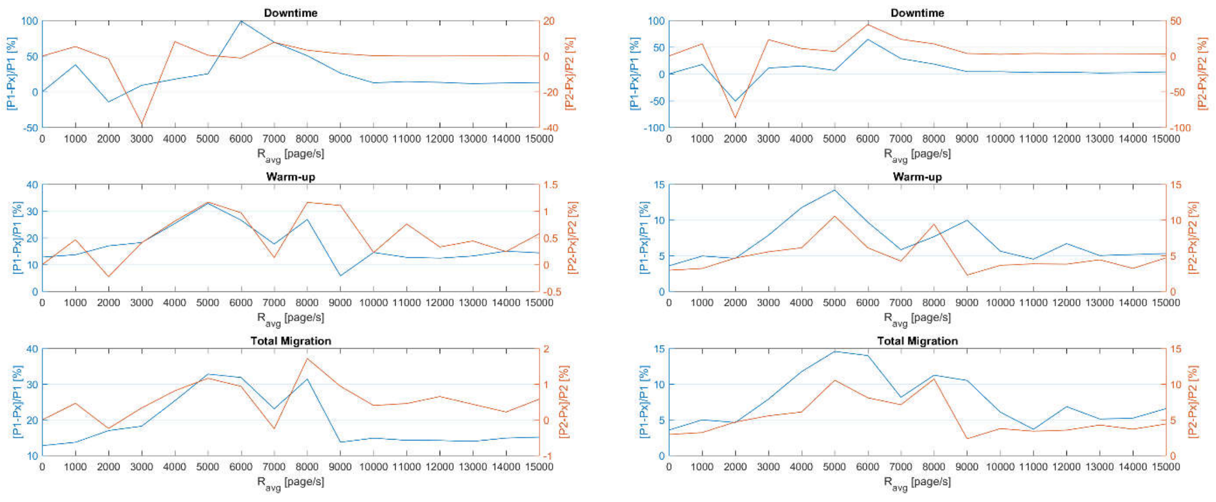

Figura 37 - -(a) P1(L)-P2(H-M) - (b) P1(L)-P2(M-M) - Pre-copy Performance - Scenario 1

Figura 38 mostra i primi due test case per lo scenario 2. Figura 38(a), il cui W_{AVBL} pattern è: 'Medium-High-High', mostra che le pagine di memoria seguono il percorso $P1$, sebbene $P2$ abbia raddoppiato la $W_{\max(P2_L1)}$. Inoltre, il miglioramento rispetto a $P2$ è quasi uguale a quello del medesimo test case scenario 1 nel quale però la W_{max} era più bassa in $P2$. Ciò implica che, sebbene in questo scenario la W_{max} è più alta dello scenario 1 SDN-C continua ad inviare le pagine di memoria lungo il percorso $P1$, poiché la combinazione d'informazioni traffico di rete e percorsi più brevi fa selezionare il percorso $P1$ e pur incrementando la W_{max} lungo $P2$ non si ottiene un significato miglioramento del '*Total Migration Time*'.

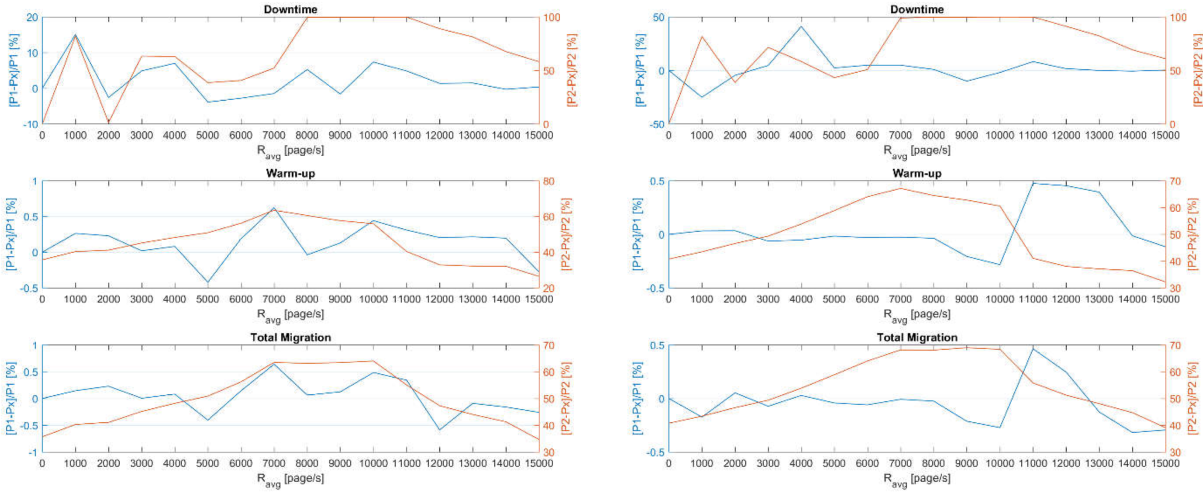

Figura 38 - (a) P1(M)-P2(H-H) - (b) P1(M)-P2(M-H) - Pre-copy Performance - Scenario 2

In Figura 38(b) il test case, in cui . W_{AVBL} è: 'Medium-Medium-High', mostra un miglioramento del '*Total Migration Time*' rispetto ad un instradamento statico verso $P2$ tra il $40\% - 70\%$ che è maggiore rispetto al test case precedente sebbene quest'ultimo abbia una globale W_{AVBL} più bassa.

Ancora una volta, evidenziamo il fatto che tramite il routing dinamico si seleziona il percorso più veloce in ogni istante. In Figura 39(a), il cui pattern W_{AVBL} è 'Low-Medium-high' decrementiamo

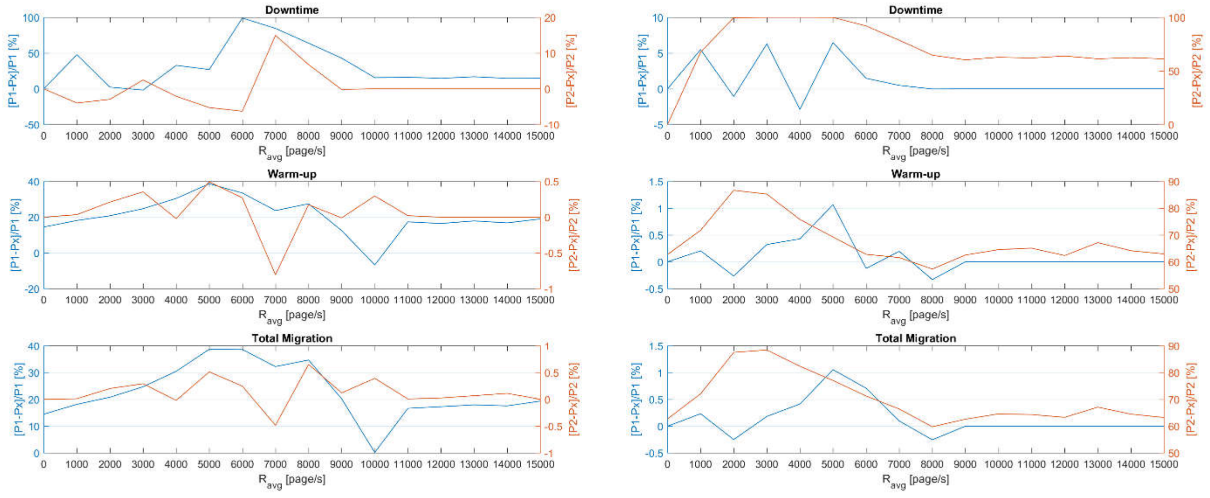

Figura 39 - (a) P1(L)-P2(M-H) - (b) P1(L)-P2(M-M) - Pre-copy Performance - Scenario 2

ulteriormente la banda disponibile lungo $P1$ lasciandola inalterata in $P2$. A causa del

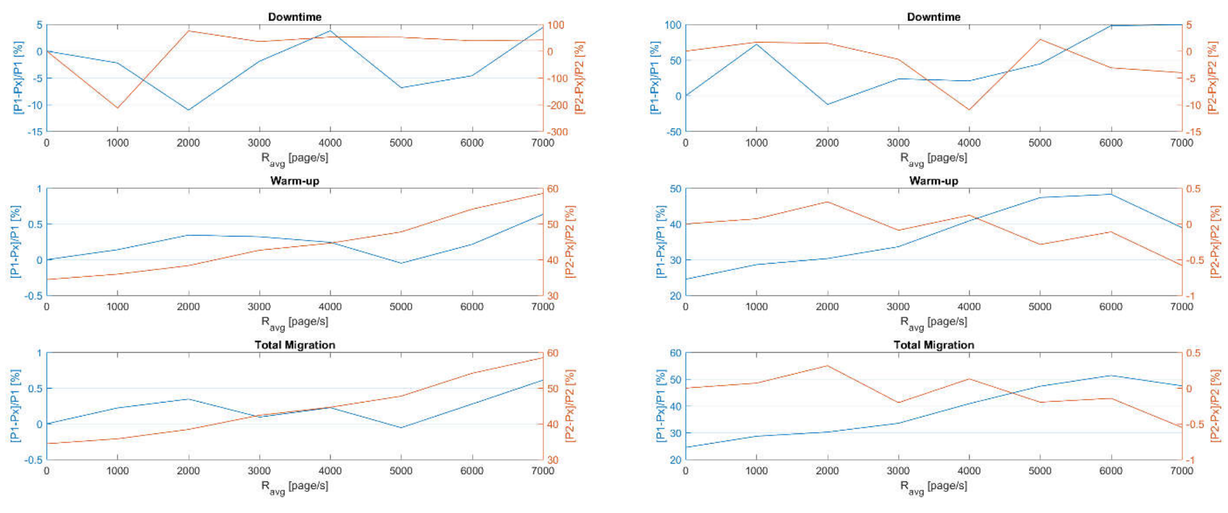

Figura 40 - (a) P1(M)-P2(H-H) - (b) P1(L)-P2(H-H) - Pre-copy Performance - Scenario 3

peggioramento delle condizioni lungo $P1$ SDN-C invia le pagine di memoria principalmente lungo il percorso $P2$. Il guadagno dal punto di vista temporale rispetto a $P1$ si attesta tra il $15\% - 40\%$. Infine Figura 39(b), in cui il W_{AVBL} pattern è: 'Low-Medium-Medium'. Decrementiamo la W_{AVBL} lungo $P2$ e manteniamo stabile e bassa quella in $P1$. Con le attuali condizioni di rete SDN-C decide di trasferire le pagine di memoria verso $P1$. Questo conferma che al variare di W_{AVBL} varia anche il routing dinamico. I risultati evidenziano un miglioramento del '*Total Migration Time*' tra il $63\% - 88\%$. I successivi scenari differiscono dai precedenti per la dimensione della RAM dell'applicazione che è adesso $32\ MB$ mentre lasciamo inviarata la W_{max}. Per quanto riguardano i test case, quelli dello scenario 3 sono identici a quelli dello scenario 1, mentre differiscono tra quello dello scenario 4 e 2. Scopo di questi ulteriori scenario è capire un eventuale o meno influenza nelle scelte fatte da SDN-C al variare della dimensione della RAM. Figura 40(a), il cui

W_{AVBL} è: 'Medium-High-High' mostra un aumento lineare del '*Total Migration Time*' rispetto il percorso $P2$ all'aumentare della frequenza media di sporcamento delle pagine di memoria R_{avg}. In dettaglio, otteniamo un miglioramento rispetto $P2$ compreso tra $35\% - 60\%$ e praticamente nullo lungo $P1$. Se confrontiamo questi risultati con quelli di Figura 36(a), dove l'unica differenza è la dimensione della RAM, possiamo concludere che la dimensione della RAM, non influenza le prestazioni di SDN-C. Figura 40(b), il cui pattern W_{AVBL} è: 'Low-High-High' è anche identico a quello in Figura 36(b). I risultati, ancora una volta, confermano la precedente conclusione che la dimensione della RAM non influenza le prestazioni di SDN-C. Infatti abbiamo un uguale miglioramento del '*Total Migration Time*' tra il $25\% - 50\%$ rispetto a $P1$. In Figura 41(a), il cui W_{AVBL} pattern è: 'Low-High-Medium' sebbene ci sia un peggioramento lungo $P2$ SDN-C preferisce inviare le pagine di memoria lungo proprio $P2$ per poter garantire una migrazione che sia la più veloce possibile. Si ottiene un miglioramento generale del *'Total Migration Time*' rispetto a $P1$ compreso tra il $10\% - 30\%$. Infine Figura 41(b), il cui W_{AVBL} pattern è: 'Low-Medium-Medium', come nel test case di Figura 37(b) si nota il reale vantaggio di utilizzare un instradamento dinamico non appena su entrambi i percorsi la W_{AVBL} è bassa. Infatti otteniamo un miglioramento del '*Total Migration Time'* tra il $4\% - 15\%$ rispetto a $P1$ e tra il $3\% - 7.5\%$ rispetto a $P2$. Quest'ultimi risultati sono fondamentalmente uguali a quelli ottenuti in Figura 37(b) confermano il fatto che la dimensione della RAM non influenza in modo significativo le prestazioni del SDN-C. Alla luce di tale osservazione per l'ultimo scenario 4, piuttosto che analizzare i medesimi casi di test come quelli in scenario 2. Abbiamo deciso di applicare nuovi pattern di W_{AVBL}.

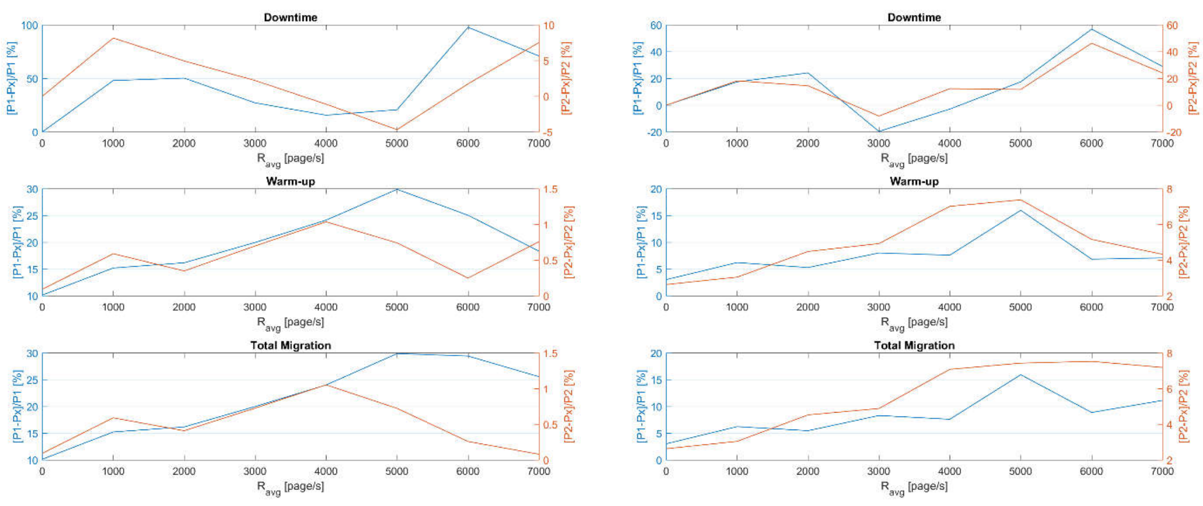

Figura 41 -(a) P1(L)-P2(H-M) - (b) P1(L)-P2(M-M) - Pre-copy Performance - Scenario 3

Nello specifico Figura 42(b) e Figura 43(b) introducono due nuovi pattern, mentre Figura 42(a) e Figura 43(a) sono uguali a quelli già analizzati, ma con differente dimensione della RAM. Nel primo test case Figura 42(a), il cui pattern è: 'Medium-High-High', otteniamo lo stesso andamento avuto in Figura 38(a) nel quale SDN-C invia le pagine di memoria lungo $P1$, anche se esso ha una minore W_{AVBL} di $P2$, ma in compenso è il percorso fisicamente più breve verso la destinazione. Il miglioramento generale del '*Total Migration Time'* rispetto a $P2$ è tra il $5\% - 20\%$. Figura 42(b). è il primo test case dove applichiamo un nuovo pattern W_{AVBL} che è: 'Low-High-Medium'. Nonostante riduciamo W_{AVBL} su entrambi i percorsi ed in particolare essa risulta molto bassa in $P1$,

SDN-C decide di inviare le pagine di memoria lungo il percorso che può garantire il più breve '*Total Migration Time'*. In dettaglio, possiamo apprezzare un miglioramento del '*Total Migration Time'* rispetto all'instradamento statico lungo $P2$ tra il $65\% - 90\%$.

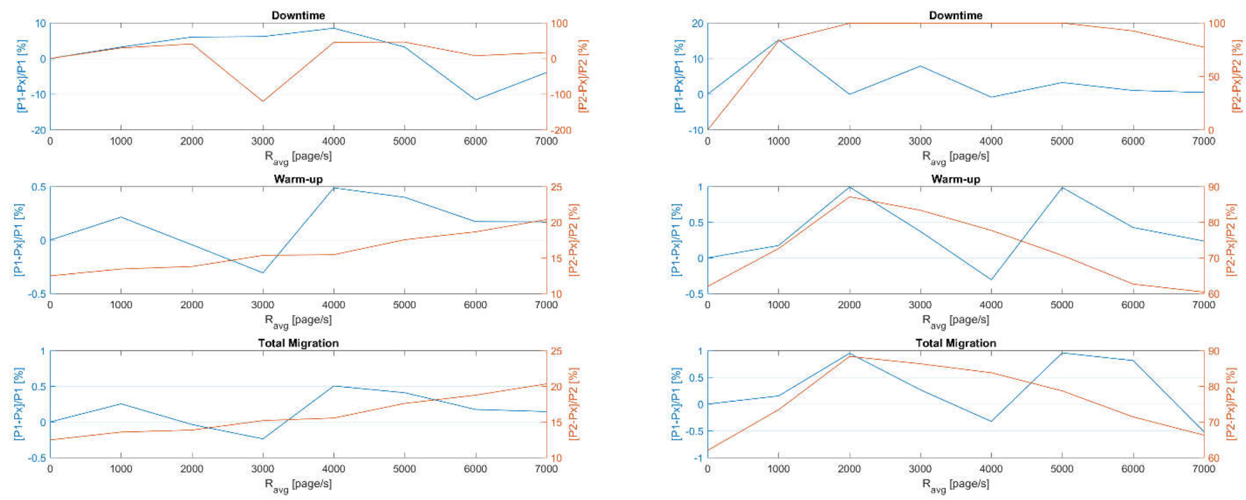

Figura 42 - (a) P1(M)-P2(H-H) - (b) P1(L)-P2(H-M) - Pre-copy Performance - Scenario 4

Figura 43 (a) mostra il terzo test case il cui W_{AVBL} pattern è: 'Low-Medium-Medium' come nel test case Figura 39(b). Come previsto, essendo la dimensione della RAM ininfluente ai fini delle prestazioni del SDN-C, otteniamo gli stessi risultati dello scenario Figura 39(b). Ovvero, continuando a decrementare la W_{AVBL} lungo $P2$ le pagine di memoria tendono ad andare verso $P1$, sebbene la sua W_{AVBL} è minore di quella di $P2$. I risultati mostrano un miglioramento del '*Total Migration Time'* rispetto all'instradamento statico lungo $P2$ tra $63\% - 88\%$.

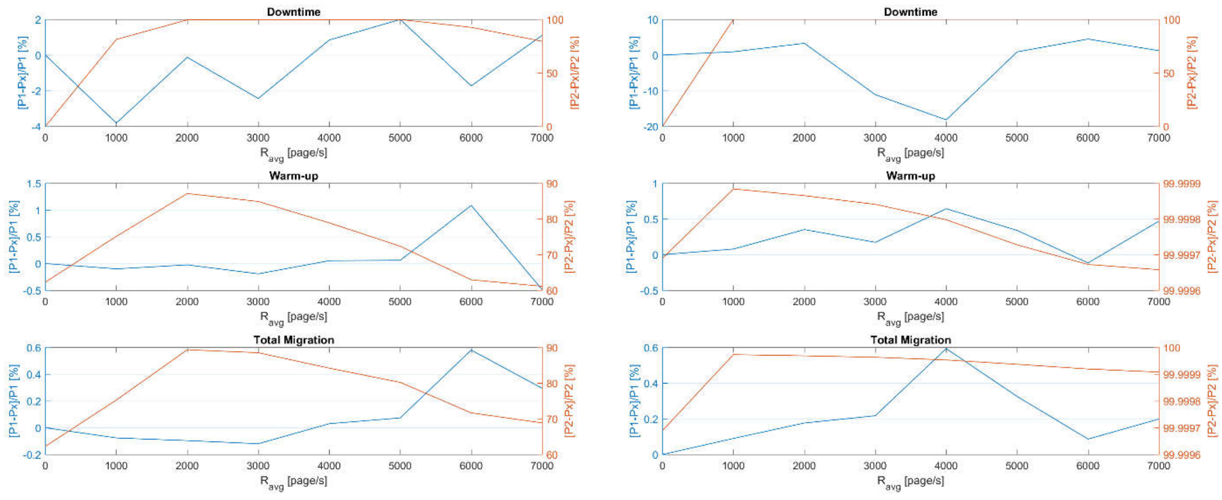

Figura 43 - (a) P1(L)-P2(M-M) - (b) P1(L)-P2(H-L) - Pre-copy Performance - Scenario 4

Infine Figura 43(b) mostra l'ultimo test case dello scenario 4, il cui W_{AVBL} pattern è: 'Low-High-Low'. Obiettivo di quest'ultimo test è vedere come reagisce SDN-C reagisce quando un percorso ha entrambe le tratte sbilanciate in termini di W_{AVBL}. Quello che notiamo è che le pagine di memoria continuano ad essere indirizzate verso $P1$, anche se la sua W_{AVBL} è bassa. Il miglioramento del '*Total Migration Time'* rispetto l'instradamento statico lungo $P2$ è intorno il 99%. Questo risultato è simile a quello ottenuto in Figura 43(a). Il motivo è che W_{AVBL} lungo il percorso $P2$ è in media uguale. Infatti, se guardiamo il W_{AVBL} pattern per $P2$ del caso di Figura 43(a) era 'Medium-Medium', mentre adesso sempre su $P2$ è 'High-Low'. Concludiamo questa lunga analisi evidenziando che anche i tempi di 'downtime', essendo dato dal rapporto tra le rimanenti pagine di memorie che si sono sporcate nell'ultimo step d'iterazione, cioè alla fine della fase di warm-up, e la W_{AVBL} beneficiano per determinati valori di R_{avg} dei miglioramenti temporali.

5.5 RIEPILOGO

Mediante questo lavoro abbiamo disegnato e modellato una nuova potenziale architettura MEC denominata '*RaTAMCA*', ed analizzato le sue prestazioni nell'ipotesi di una migrazione svolta secondo l'algoritmo di pre-copy. L'unione di tecnologie quali MEC e SDN, nell'ipotesi di migrazione di applicazioni RAM-intensive, hanno mostrato che la selezione dinamica del percorso migliore ad ogni iterazione della fase di warm-up dell'algoritmo di pre-copy è migliore di una scelta puramente statica. Dato che l'abbassamento dei tempi di migrazione (warm-up e downtime) rende la migrazione più fluida, è chiaro che con l'introduzione del supporto SDN all'algoritmo di pre-copy può aprire la strada verso una migrazione del servizio non solo robusta, ma più efficiente. In particolare, abbiamo prima progettato una nuova architettura MEC, poi modellato il traffico di rete e simulato un controller SDN applicato durante l'algoritmo di pre-copy. I risultati hanno evidenziato un miglioramento del 'Total Migration Time' e anche del 'Downtime' con punte oltre il 90%. Un'altra considerazione derivata dall'analisi è che la dimensione della RAM non influenza le prestazioni del controller. Un potenziale lavoro sarebbe quello di sviluppare un modello di sporcamento delle pagine di memoria più complesso di quello attuale così da poter validare meglio questi risultati.

6 KATA CONTAINERS: UN' ARCHITETTURA EMERGENTE PER ABILITARE IN MODALITÀ VELOCE E SICURA I SERVIZI MEC

Le future applicazioni saranno soltanto possibili mediante una rete di Mobile Edge Servers collocati in prossimità dell'utente mobile. Data sia la mobilità dell'utente che il variabile carico di lavoro sul server la migrazione del servizio sarà un aspetto fondamentale da tenere in conto. Per tali motivi, un'architettura standardizzata ed aperta dovrebbe essere progettata per realizzare una service migration in maniera sicura ed entro i vincoli temporali del servizio richiesto. La maggior parte delle ricerche svolte sino ad oggi si sono focalizzate sull'uso delle virtual machine (VM) o dei containers o impiegando entrambe. Una soluzione finale potrebbe essere un'architettura che abbia i vantaggi di entrambe le tecnologie come la sicurezza delle VM e la velocità di esecuzione dei containers. Le soluzioni '*custom*' necessitano una continua ottimizzazione da caso a caso e soprattutto non sono conformi con uno standard. A tale scopo, presentiamo una nuova architettura, Kata Containers, che è supportata da Open Stack Foundation (OSF), e supporta standard industriali per le immagini come OCI, interfaccia CRI di Kubernetes come pure tecnologie legacy di virtualizzazione.

6.1 INTRODUZIONE

Secondo Cisco Global Cloud Index (69) entro il 2021 nel segmento consumer, lo streaming video e social networking rappresenteranno le applicazioni col maggiore tasso di crescita e dunque, un esplosione in mobilità di applicazioni content-based. Di conseguenza, una migrazione del servizio, conosciuta anche come '*Live Migration*' giocherà un ruolo importante. Molte soluzioni sono, sino ad oggi, state focalizzate sulle VM o containers. VM è ormai una tecnologia molto consolidata che assicura robustezza e sicurezza, ma al contempo necessita fare un intero snapshot dell'intera VM per replicare il servizio dando luogo ad una lenta migrazione, elevato consumo di storage e anche di larghezza di banda. Siccome le nuove applicazioni richiederanno stringenti vincoli in termini di ritardo, tempi di risposta e jitter, gli ultimi studi hanno presentato una soluzione basata sui containers che hanno tempi di esecuzione molto più brevi delle VMs e anche minore overhead. La piattaforma di gestione container, ad oggi, più utilizzata è Docker (70) (71) che è diventata uno '*standard de facto*'. Nonostante questi evidenti benefici nell'impiego dei containers bisogna evidenziare che un recentissimo bollettino sulla sicurezza da '*Common Vulnerabilities and Exposures*' (CVE), datato Marzo 2019, ha annunciato un bug su Docker che permette ad un container malevolo di sovrascrivere il binario runc e ottenere accesso root per poi eseguire codice nell'host (72). Le cause di tale bug sono dovute ad una mancata gestione del file-descriptor in runc. La gravità di tale bug è che può anche dirottare altri sistemi containers come LXC (73) e Apache Mesos (74). In altre parole, la maggior parte, se non tutti, dei sistemi di cloud container sono vulnerabili a questo attacco. Questa debolezza ci fa capire che il solo utilizzo dei containers non può essere la soluzione per eseguire le future applicazioni MEC, e dunque, urge focalizzarsi su un'architettura che sia più complessa e al contempo standardizzata. A tal proposito un recente contributo dato da (75) sfrutta l'architettura di storage a livelli di Docker per ridurre la quantità di dati da trasferire. Un passaggio chiave di tale lavoro è proporre un sistema di migrazione con due livelli di isolamento per una migliore sicurezza durante l'operazione di offloading. Gli autori in (75) evidenziano l'importanza di applicare un approccio di sicurezza a due livelli, secondo il modello '*defence-in-depth*' (76). Questo modello di sicurezza è dato dalla combinazione di VMs e containers nei quali i servizi sono per prima isolati ed in esecuzione dentro il loro rispettivo container e successivamente ulteriormente isolati dentro differenti VMs. Questa soluzione apre le porte a valutare nuove tecnologie che potrebbero assicurare sia performance che sicurezza nella live

migration e non solamente. In tale contesto, una nuova ed emergente tecnologia, di nome '*Kata Containers*' (77), potrebbe colmare il gap tra sicurezza e velocità e gettare le fondamenta per una soluzione open e standardizzata piuttosto che custom. Al meglio della nostra conoscenza, questo lavoro è il primo che introduce questa architettura nei suoi aspetti chiave e li confronta con quelli di Docker. I contributi apportati da tale lavoro di ricerca sono:

1. Introduzione dell'architettura Kata Containers allo scopo di farla conoscere il più possibile alla comunità scientifica
2. Analisi qualitativa per indirizzare le future ricerche su quali funzionalità bisognerà sviluppare o migliorare

Infine questo lavoro di ricerca è stato presentato alla conferenza di Granada: '*The 6th IEEE International Conference on Internet of Things: Systems, Management and Security (IOTSMS 2019)*'

6.2 Kata Containers

In questa sezione descriviamo diversi aspetti architetturali di Kata Containers quali networking, storage, devices ed interfacce.

6.2.1 Architettura

Il progetto Kata Containers nasce dalla fusione di altri due progetti, '*Intel Clear Containers*' e '*Hyper.sh runV*' nel Dicembre 2017 con lo scopo di costruire VMs estremamente leggere dalle prestazioni simili a quelle dei containers, e al contempo fornire l'isolamento del workload e quella sicurezza aggiuntiva fornita da un secondo livello di difesa che è quello delle VMs. La prima versione 1.0 di Kata Containers fu rilasciata nel Maggio 2018. Nel momento in cui scriviamo la versione attuale è la 1.7.0. Essendo Kata Containers un container runtime OCI-compliant, la sua integrazione con tutte le piattaforme di gestione compatibili con OCI è del tutto trasparente. L'integrazione consiste semplicemente nel rimpiazzare il container runtime di Docker, cioè runc, con quella di Kata. Figura 44 mostra quanto facile sia integrare Kata Containers in piattaforme che gestiscono il formato OCI o interfacce CRI o tecnologie legacy di virtualizzazione.

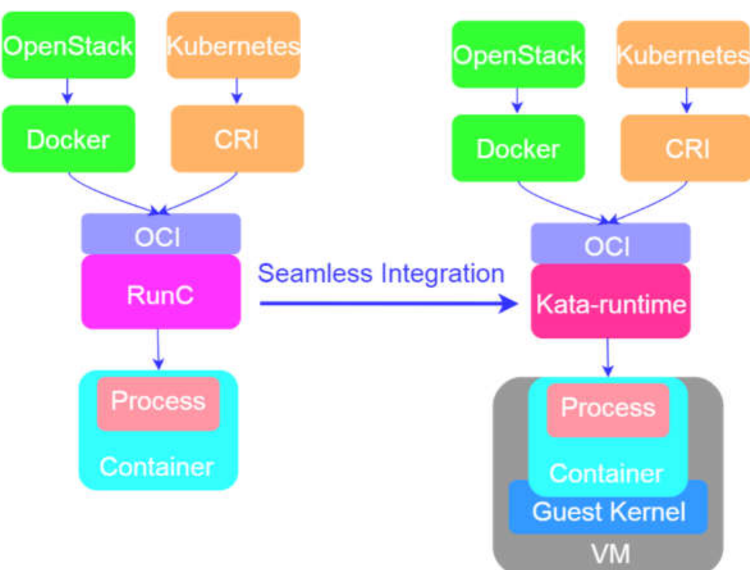

Figura 44 - Integrazione Kata Containers

L’architettura software include quattro componenti principali:

1. **Kata-runtime:** è un container runtime compatibile con il formato OCI e ha il compito di gestire i comandi specificati dalle specifiche OCI runtime, per esempio, invocazione dell’hypervisor per creare una VM leggera e veloce per ogni container o pod ed eseguire istanze di kata-shim
2. **Kata-shim:** un processo in esecuzione nel sistema host per gestire input/output streams di tutti i containers. Esso agisce come se fosse il processo container (in realtà esso è in esecuzione nella VM) così da essere visibile al processo ‘reaper’, che si trova nell’host, e può, dunque, monitorare e controllare il processo container. Questo componente è obbligatorio per essere OCI-compliant.
3. **Kata-proxy:** è un processo che offre l’accesso al Kata-agent presente nella VM a tutti i processi Kata-shim e Kata-runtime clients associati a quella VM. Il principale compito è quello di instradare tutto input/output ed i segnali che scambiano le istanze Kata-shim e il Kata-agent. Il protocollo di comunicazione è gRPC (78) e tutte le richiese sono multiplexate da yamux (79)
4. **Kata-agent:** è un processo in esecuzione nel guest dentro la VM. Il suo compito è configurare l’ambiente per gestire i containers e i processi in esecuzione dentro i containers.

L’attuale architettura software è semplificata con la fusione di tre componenti, quali Kata-shim, Kata-proxy e Kata-runtime in uno soltanto di nome ‘*containerd-shim-kata-v2*’. L’interfaccia di comunicazione tra containerd-shim-kata-v2, hypervisor e Kata-agent è ‘*vsock*’, mentre il protocollo di comunicazione resta gRPC. Questa soluzione permette di evitare l’uso del Kata-proxy poiché l’interfaccia vsock esegue il multiplex di tutte le richieste gRPCs. Figura 45 mostra l’architettura software di Kata Containers.

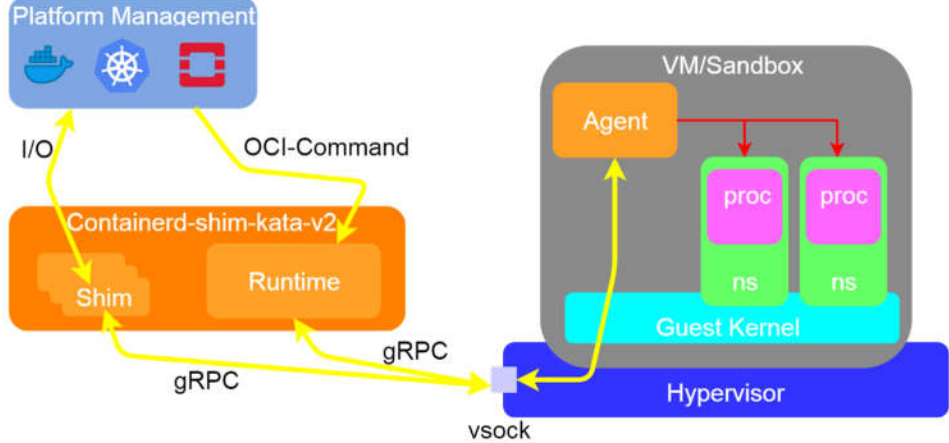

Figura 45 - Architettura Software Kata Containers

Per una maggiore comprensione su come i componenti di Kata interagiscono tra loro, supponiamo di creare un nuovo container.

Figura 46 mostra i messaggi scambiati durante la creazione di un Kata containers. Alla ricezione del

Figura 46 - Kata Containers Creation

comando '*create*' da una piattaforma di management (come per esempio Docker, Kubernetes), ① Kata-runtime chiama una procedura di '*virtocontainers*', che è una generica ed agnostica specifica di runtime, ovvero una libreria per la virtualizzazione hardware per creare delle '*sandbox*', che rappresentano un ambiente dove un insieme di containers condividono lo stesso 'networking namespace' e storage. In dettaglio, per conto di Kata-runtime creerà ed inizierà la configurazione di rete tramite alcune operazioni come: ②③ in cui si crea il network namespace del guest (guest netns), la coppia di interfacce '*veth*' per collegare i due network namespace, l'interfaccia '*TAP'*, necessaria per collegare la VM con l'interfaccia '*veth*' interna al 'guest netns' mediante connesione *'MACVTAP'*. Successivamente, una chiamata all'hypervisor ④ e ⑤ avvierà la VM nel 'guest netns' fornendo l'interfaccia di comunicazione 'TAP'. Prossimo step è l'avvio del componente Kata-proxy, che si connetterà all'appena creata VM. Come descritto prima, il principale compito di Kata-proxy è mettere in comunicazioni i componenti di Kata presenti nel 'host netns' con la VM nel 'guest netns'. Prossimo step è la creazione del container o POD dentro la VM ⑦ e ⑧. Questa operazione è svolta dal Kata-agent che configura l'ambiente sulla base del file di configurazione e deposita il processo container. Infine, Kata-runtime fa partire Kata-shim ⑨ che si connetterà al gRPC server per potere direttamente controllare lo stato del processo container direttamente dall'ambiente host. Kata-shim è un componente obbligatorio ed è in rapporto 1: 1 coi processi containers. Il motivo per cui è obbligatorio è perché Kata-runtime è al contrario un componente temporaneo, ovvero dopo aver creato e avviato il processo container esso termina ed esce. Come anticipato precedentemente, una tra le più importanti caratteristiche di Kata Containers è l'introduzione del doppio livello di sicurezza in cui ogni processo container è isolato all'interno di una VM alleggerita ed ottimizzata il cui guest kernel è isolato sia dal kernel host che da altri containers a loro volta isolati nello proprie VMs. Quindi, ci chiediamo:

‘***Come è possibile ottenere delle VMs così veloci e leggere?***’ Sono fatte diverse ottimizzazioni che non riguardano soltanto l’hypervisor, ma anche la VM e le comunicazioni tra la stessa VM e i devices al fine di ottenere prestazioni in linea coi tempi dei containers. Per quanto riguarda l’hypervisor, l’architettura di Kata può supportare diversi hypervisors e di default utilizza QEMU/KVM. Più precisamente, utilizza una versione ottimizzata di QEMU di nome, qemu-lite (80) per velocizzare il tempo di boot e ridurre l’utilizzo della memoria (RAM). Ciò è reso possibile grazie alle seguenti ottimizzazioni:

1. **Machine accelerators (MA):** sono architetture specifiche per ottimizzare le prestazioni. Una di queste è NVDIMM (81) che si basa sull’architettura x86 in modalità di accesso diretto allo storage (DAX). Essa permette la condivisione del file rootfs dell’host in modalità di sola lettura come un dispositivo di memoria persistente per le VMs.
2. **Kernel same-page merging (KSM):** è una funzione di KVM che permette di condividere pagine di memoria identiche tra differenti VMs, con la finalità di deduplicare la memoria e massimizzare la densità del numero di containers nell’host.
3. **Hot plug devices (HPD):** La VM è avviata con una minima quantità di risorse per velocizzare il boot time ed utilizzare una minima quantità di memoria. Una volta avviata, non appena necessita più risorse, l’hypervisor automicamente in modalità hotplug aggiunge risorse al volo.
4. **Fast Template (FT):** Un insieme di template preconfigurati di VMs ultra leggere da mandare in esecuzione molto velocemente.

Le ottimazzioni lato VM sono:

1. **Guest Kernel minimal (GKM):** È un kernel altamente ottimizzato per ridurre al minimo il boot time e l’utilizzo della memoria, fornendo solamente quei servizi strettamente indispensabili per un container worload.
2. **Guest Image (GI):** È un sistema operativo minimale (mini O/S) per avere bootstrap velocissimi. Ciò è reso possibile poiché il mini O/S ha soltanto due servizi in esecuzione all’avvio; systemd e Kata-agent. Il primo manda in esecuzione Kata-agent, mentre il secondo crea l’ambiente per il container.

Infine la comunicazione tra le VMs e i dispositivi è anche ottimizzata tramite una tecnica nota come hardware passthrough (HWP) che dà alla VM accesso diretto ai dispositivi dell’host, come per esempio, scheda di rete pci, ottenendo quindi prestazioni vicine a quelle native. Altre tipologie di comunicazione sono single root input/output virtualization (SR-IOV) (82) che permette ad un dispositivo fisico PCIe di apparire come se vi fossero multipli dispositivi fisici PCIe separati e la VirtIO interface (83) che permette alle VMs un accesso semplificato a dispositivi virtuali quali dispositivi a blocchi, adattatori di rete e console. Figura 47 sintetizza le varie ottimizzazioni che rendono possibile una VM leggera e veloce come un container.

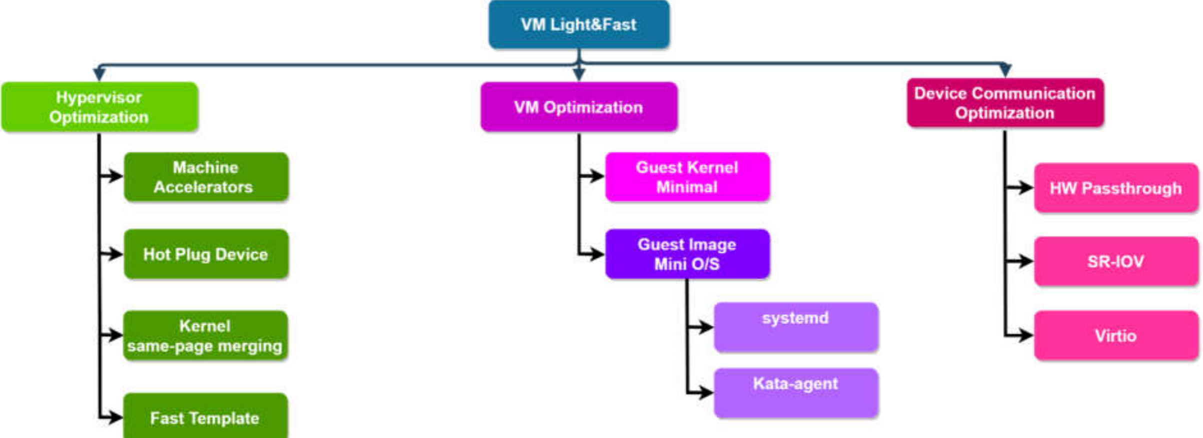

Figura 47 - Ottimizzazioni di Kata Containers

6.2.2 NETWORKING

Per quanto riguarda la rete, il container engine, situato presso il sistema host, configura il '*networking namespace*' per ogni VM. Tale networking namespace è isolato da quello del sistema host, ma in condivisione con tutti i containers. La comunicazione tra host e container namespace avviene per mezzo di una coppia di virtual ethernet (veth). Una si trova nel namespace dell'host ed un'altra in quello del container. Questa modalità di comunicazione è tipica dei containers, ma alcuni hypervisors, come QEMU, possono soltanto gestire interfacce differenti, di nome '*TAP*', per la comunicazione con le VMs. Quindi vi sarebbe un'impossibilità di comunicazione coi cointainer visto che sono inglobati in una VM. La comunicazione è comunque possibile tramite il driver '*MACVTAP*' (84) che mette in comunicazione le interfacce veth e TAP. È anche possibile monitorare il traffico di rete tramite un modulo piazzato le due interfacce TAP e veth. Infine, Kata containers supporta due modelli di networking che sono: container networking model (CNM) adottato da Docker e container networking interface (CNI) adottato da Kubernetes e Podman. Entrambi i modelli danno la possibilità di selezionare uno specifico tipo di container networking, associarsi ad una o più reti e utilizzare più drivers di rete contemporaneamente. Figura 48 sintetizza le connessioni di rete tra host e il network container namespace (tramite veth) e tra host network namespace e container (tramite MACVTAP).

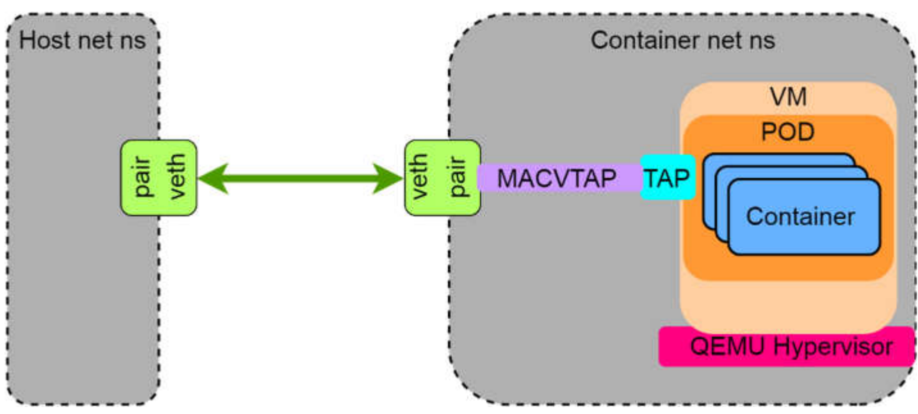

Figura 48 - Kata Networking

6.2.3 STORAGE

Kata containers può gestire lo storage sia a livello file che a livello blocchi. Nel primo caso, fa uso del file system '*9pfs*' (85) che permette di montare l'immagine rootfs in modalità cold-plug, mentre a livello di dispositivo a blocchi, può montare interi volumi in modalità hot-plug usando il driver '*device-mapper*' (86). Tale driver ha prestazioni migliori di 9pfs, ma al contempo ha qualche problema di compatibilità con le versioni recenti di Docker.

6.3 ANALISI QUALITATIVA: CONFRONTO RUNC VS KATA-RUNTIME

Dopo aver introdotto l'architettura Kata Containers e descritto le sue principali funzionalità è ora di presentare un confronto qualitativo con il container runtime di Docker che è '*runc*'. L'analisi si basa sulle attuali limitazioni e problemi ancora aperti, ma considerando anche altri aspetti già ben consolidati come la sicurezza, le perfomances, resource management e networking.

Per fare quest'analisi, definiamo una scala di qualità a 5 valori (*poor*, *satisfactory*, *good*, *very good*, *excellent*) per valutare diversi aspetti dei due container runtime. Tabella 10 mostra per ogni valutazione qualitativa la sua descrizione e il corrispondente punteggio di qualità assegnato.

Grade	Description	Quality Point
Poor	Funzionalità non ancora implementata	1
Satisfactory	Funzionalità in stato elementare necessita molti miglioramenti	2
Good	Funzionalità con diversi punti di forza, ma necessita ancora qualche miglioramento	3
Very Good	La funzionalità è quasi perfetta in ogni suo aspetto	4
Excellent	La funzionalità è al top della sua implementazione con elevate prestazioni	5

Tabella 10 - Scala di Qualità a 5 valori

Abbiamo considerato 10 aspetti per funzionalità da sviluppare, sicurezza, performance, limitazioni di rete, gestione risorse e condivisione, facilità d'integrazione nelle attuali piattaforme cloud. Il primo aspetto è stato quello di poter eseguire la **'*migrazione live*'**, '*live migration*' del workload, nel momento in cui scriviamo Kata-runtime non ha ancora implementato funzionalità quali '*checkpoint e restore*' a differenza di Docker in cui sono presenti. Per tale motivo abbiamo assegnato un punteggio di qualità pari ad 1 a Kata-runtime e 3 a runc. Per quanto riguarda il **'*modello di sicurezza*'** Kata-runtime supera runc grazie al doppio livello di sicurezza fornito che ci ha portato ad assegnare 5 a Kata-runtime e 3 a runc. Abbiamo, inoltre, considerato le ***'security options'*** come SecComp (87), AppArmor (88), SeLinux (89) che possono essere abilitate in Docker. Poiché Kata-runtime supporta, al momento, solamente (87), per tale motivo, abbiamo assegnato un punteggio pari a 2 a Kata-runtime e 4 a runc. Per quanto riguarda l'***'isolamento'*** abbiamo assegnato 5 a Kata-runtime e 3 a runc. Motivo principale è che l'architettura Kata-containers è più sicura di Docker e non permette, in caso di 'root-escalation', di compromettere anche il kernel host. Per le ***'performance'***, intese come boot time, sono quasi uguali a condizione che la VM sia avviata con le minime risorse e cioè una sola virtual cpu (VCPU) e nel momento in cui necessita maggiori risorse computazionali possono essere aggiunte in modalità hot-plug. Per tale motivo, abbiamo assegnato un punteggio pari a 5 per entrambe le architetture. L***'integrazione'*** di questi due runtime containers in altre piattaforme è ottima per entrambe visto che sono OCI-compliant. Abbiamo, dunque, assegnato per entrambe un punteggio massimo pari a 5. Abbiamo anche valutato le attuali *'limitazioni di rete'* in Kata-runtime. La prima è che Kata-runtime non supporta la possibilità di unire differenti containers namespace e, dunque, non può condividere un comune ***'network namespace'*** così come le relative interfacce presenti. Per tale motivo, abbiamo assegnato 1 a Kata-runtime e 3 a runc. La seconda limitazione è che Kata-runtime non supporta l'accesso alla configurazione di rete dell'host (***'host networking configuration'***) da dentro la VM. In realtà tale opzione in Kata-runtime sarebbe attivabile solo che l'effetto sarebbe diverso rispetto a

runc e comunque potrebbe portare alla rottura della configurazione di rete dell'host. Per tali ragioni, abbiamo assegnato a Kata-runtime 1 e 3 a runc. Per quanto riguarda ***'host resource sharing'***, runc può accedere in modo privilegiato che significa che può accedere direttamente ai dispositivi dell'host, mentre Kata-runtime, sebbene supporti anche tale modalità, il risultato è diverso cioè accede ai dispositivi della guest VM. Nonostante questa differenza sia considerata una limitazione, da un punto di vista della sicurezza è in realtà un valore aggiunto nel caso in cui si verifichi, per esempio una root escalation del container che resterebbe confinato nella guest VM e quindi i danni sarebbero limitati. Per tale motivo, abbiamo deciso di assegnare 4 a Kata-runtime e 3 a runc. Infine, abbiamo anche considerato ***'resource constraint management'*** che risulta essere molto più semplice in runc che in Kata-runtime. Infatti, runc usa soltanto il concetto dei control groups (cgroups) per limitare, assegnare priorità, controllare e monitorare le risorse, mentre per Kata-runtime, a causa del doppio livello di sicurezza, potrebbe essere necessario, per ottenere il medesimo risultato, configurare i vincoli su più livelli. Possiamo, quindi, dire che il resource constraint management è di tipo a '*grana-grossa*' in runc, mentre è a '*grana-fine*' in Kata-runtime. Per tale maggiore complessità, abbiamo assegnato 4 a runc e solo 3 a Kata-runtime. Di seguito Figura 49 sintetizza questa analisi qualitativa mediante un grafico radar.

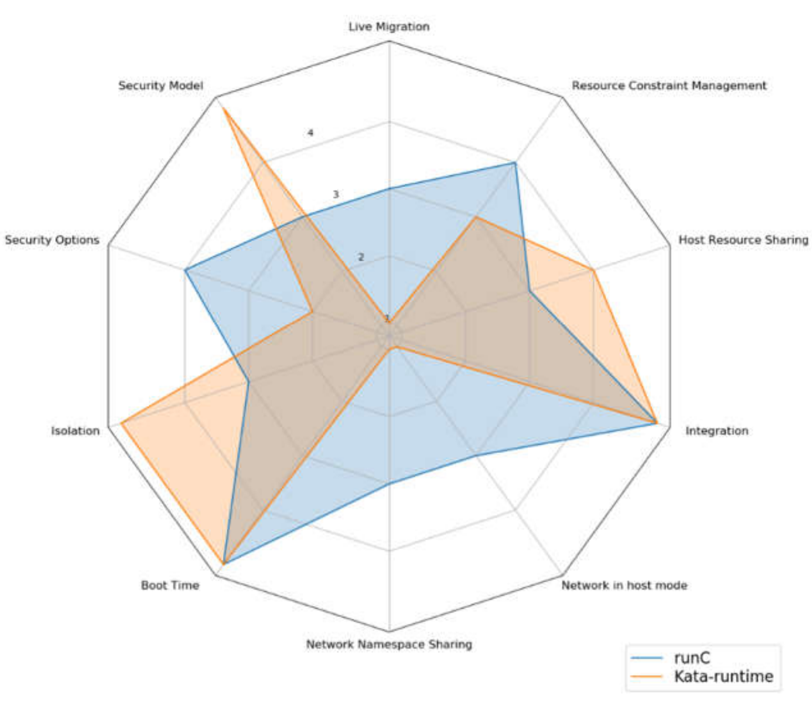

Figura 49 - Confronto Qualitativo Kata-runtime vs runc

6.4 Riepilogo

Attraverso questo lavoro, abbiamo introdotto una nuova ed emergente architettura, Kata Containers, che mette insieme i lati positivi sia delle VM che dei containers in un'unica soluzione. Infatti, tale architettura ha un modello di sicurezza molto robusto grazia alla virtualizzazione hardware tipica delle VMs e al contempo la velocità dei containers. Obiettivo di tale lavoro è stato far conoscere il più possibile alla comunità scientifica tale soluzione poiché crediamo che Kata Containers possa diventare la soluzione standard nel prossimo futuro. Come già citato precedentemente gli autori in (75) hanno rimarcato l'importanza di progettare un'architettura più robusta dal punto di vista della sicurezza al fine di garantire una corretta '*live migration*' dei servizi MEC. Infatti, se da un lato i containers forniscono quella velocità di risposta tale da poter soddisfare gli stringenti requisiti come delay, response time tipici di applicazioni di realtà aumentata, gaming online e data analytics, dall'altra abbiamo un problema di sicurezza insito nel disegno originale dei containers e cioè la condivisione del kernel host con tutti i containers. Dualmente, le VMs sono molto più sicure grazie alla virtualizzazione hardware, ma al contempo il loro overhead li rende molto lente ed inadeguate per molti servizi MEC. Ecco quindi, che Kata Containers cerca di prendere il meglio da entrambe le tecnologie per consegnare una soluzione veloce e sicura. Altri aspetti, non prettamente tecnologici che rendono Kata Containers una soluzione molto interessante è che oltre ad essere open-source è che il progetto è iniziato da OpenStack Foundation ed è attualmente supportato dalle più grandi ed importanti aziende del mondo cloud, delle infrastutture hardware e software, come per esempio aws Amazon, Microsoft, Google Cloud, Intel, Huawei, ZTE, Dell EMC e molte altre. Crediamo, dunque, che tramite la nostra analisi comparativa sia emerso lo stato dell'arte di tale soluzione e su cosa la comunità scientifica dovrà focalizzarsi. Infatti, aspetti come sicurezza, prestazioni, integrazione sono già ad un buon livello di maturità, mentre altri molto importanti quali le operazioni necessarie per eseguire 'live migration' sono ancora da implementare così come da migliorare la condivisione delle risorse. Per riassumere Figura 50 sintetizza tutte le funzionalità chiave di questa nuova soluzione.

Symbol	Feature	Description
	Security	• Double Security Layer • No hackable Kernel Host
	Integration	• Seamless integration with many OCI-compliant container management systems
	Isolation	• Each workload runs inside its own VM • Possibility to differentiate the Kernel per-workload/Tenant

	Performance	• Resource Efficiency
		  ◦ Minimal Memory Footprint
		  ◦ Optimal CPU utilization
		• I/O Optimization
		• Performance as standard Linux Container

MEC	Use Case	• Suitable for deployment within a multi-tenant untrusted environment • Potential solution for many MEC use cases where security&speediness is a must

Figura 50 - Riepilogo Features di Kata-Containers

7 CONCLUSIONI

Durante questi tre anni di studio e ricerca ci si è posti l'obiettivo di analizzare uno delle tante sfide che il 5G porterà con sè e cioè la migrazione live ('live migration)' in mobilità di workload gestiti dalla rete di server di mobile edge computing (MEC) collocati in prossimità dei client mobili. L'individuazione del problema è stata resa possibile solo dopo aver svolto una preliminare fase di studio e ricerca sul 5G ed un'altra più specifica e dettagliata sull'architettura MEC ed in particolare una piattaforma open-source di MEC di nome M-CORD. Le ragioni che mi hanno portato a focalizzarmi sugli aspetti della live migration sono stati fondamentalmente due: Il primo è che secondo le più recenti indagini condotte da Cisco i servizi più richiesti e che determineranno un incremento esponenziale di flusso dei dati saranno quelli di contenuti audio/video streaming in mobilità. Seconda motivazione è che vi saranno anche tanti altri nuovi servizi, non così pervasivi, ma comunque importanti quali realtà aumentata, data analytics i cui requisiti temporali imporranno delle migrazioni veloci dell'ordine di alcuni millisecondi e soprattutto trasparenti per gli utenti. Se a tutto ciò si aggiunge che la mobilità è una caratteristica intrinseca di qualunque futura rete radiomobile si deduce che l'unione di nuovi servizi con stringenti requisiti temporali e mobilità fanno sì che la live migration sia un obiettivo prioritario. Sulla base ciò, un primo lavoro è stato quello di analizzare il comportamento di uno tra i più robusti algoritmi che possono essere utilizzati nella migrazione dei servizi che è il pre-copy. In particolare, ho proposto la modifica di un esistente framework di migrazione rimpiazzando l'algoritmo di post-copy con quello di pre-copy nel caso in cui le applicazioni da migrare fossero di tipo RAM-intensive. Dalle simulazioni condotte ho evidenziato come il service downtime e più in generale l'algoritmo è molto influenzato dalle scelte dei valori di soglia dei suoi parametri e dunque una loro pre-ottimizzazione porterebbe a reali vantaggi in termini di un minore service downtime. Altra osservazione, in considerazione del fatto che la banda disponibile è variabile e non costante ed in funzione del traffico di rete, è che l'integrazione di un controller che abbia una visione globale della topologia logica della rete possa ulteriormente i tempi di migrazione. Nel capitolo (RICONOSCIMENTO RISOLUZIONE VIDEO VIA METRICHE DI MEMORIA) si è invece voluto dimostrare come in uno scenario di content delivery l'intelligenza artificiale ed in particolare machine learning possa garantire agli abbonati il monitoraggio e di conseguenza il rispetto degli accordi contrattuali (SLA) tra il content provider e l'abbonato mobile. In un ipotetico scenario di abbonamento per un servizio di video streaming tra utente e content provider ho progettato e sviluppato una rete neurale capace, che dopo un opportuno training iniziale, di discernere in tempo reale la qualità del contenuto multimediale così da inviare un'opportuna segnalazione tramite la rete di violazione del SLA e indurre una migrazione del servizio verso un altro MEC server le cui condizioni siano tali da poter garantire la qualità pattuita. I risultati hanno mostrato che il nostro modello di risoluzione video ha raggiunto ottimi risultati di accuratezza su dati mai visti pari a circa il 95%. Questo ha dimostrato come AI può agevolare l'ottimizzazione dei servizi ed in particolare determinare un preciso istante in cui migrare il servizio. Nel capitolo (OTTIMIZZAZIONE PRE-COPY: ARCHITETTURA RATAMCA) ho esteso il primo lavoro di analisi dei tempi migrazione proponendo una nuova architettura in cui la presenza di un controller SDN collocato in concomitanza col MEC controller può ottimizzare durante ogni fase dell'algoritmo di pre-copy i tempi di migrazione delle pagine di memoria. In particolare si sono ipotizzati dei modelli di traffico così da considerare vari scenari di analisi con larghezza di banda variabile nel tempo. I risultati hanno confermato le nostre ipotesi che l'instradamento dinamico può ulteriormente migliorare la migrazione del servizio. Ulteriore osservazione è stata quella che la dimensione della RAM dell'applicazione non influenza nel complesso le prestazioni del controller. Infine nel capitolo (KATA CONTAINERS: UN’ ARCHITETTURA EMERGENTE PER

ABILITARE IN MODALITÀ VELOCE E SICURA I SERVIZI MEC) ho introdotto una nuova architettura che potrebbe essere considerata nel prossimo futuro l'architettura di riferimento per la migrazione live dei servizi. Infatti le ultime ricerche si sono indirizzate verso una soluzione mista comprendente sia le VMs che i containers, visto che in alcuni aspetti queste due tecnologie si dimostrano essere complementari. Il problema però di queste ultime ricerche è che essendo delle soluzioni ad hoc portano con sé tutti i relativi limiti di diffusione, uno su tutti non essere open e standardizzata. Invece questa nuova soluzione il cui nome è Kata-containers si rifà a standard aperti e soprattutto compliant col formato open dei containers che fa sì che possa essere facilmente integrata nelle attuali piattaforme di orchestrazione e management. Essendo un progetto molto nuovo, lo scopo è stato di condurre un'analisi dettagliata della sua architettura e delle sue caratteristiche funzionali e fare un confronto qualitativo con uno dei più diffusi containers runtime che è runc. Attraverso tale analisi si sono, pertanto, perseguiti due obiettivi diffondere l'esistenza di questa nuova soluzione e al contempo incentivare la comunità scientifica ad investire su essa andando a sviluppare alcune funzionalità ancora non presenti. Una su tutti quella della migrazione del workload che è condizione necessaria per la realizzazione della live migration. Ricapitolando, questo viaggio fatto di studio e ricerca ha portato a concludere che i futuri servizi MEC dovranno puntare ad un algoritmo robusto come quello di pre-copy , tener conto della dinamicità dell'ambiente grazie l'ausilio di SDN e di tecniche di machine learning per garantire il rispetto della qualità del servizio e soprattutto definire un'unica open-source e standardizzata architettura abilitante la live migrazione nel rispetto dei requisiti del servizio richiesto.

PUBBLICAZIONI

- Title: ‘**Performance Analysis of Memory Cloning Solutions in Mobile Edge Computing**’ Publication date: **Oct 15, 2018** Publication description: **Fifth International Conference on Internet of Things: Systems, Management and Security/IEEE**
- Title: ‘**Recognizing Video Resolution by Monitoring Memory Metrics in Mobile Clients**’ Publication date: **Jun 10, 2019** Publication description: **The Fourth IEEE International Conference on Fog and Mobile Edge Computing (FMEC 2019)**
- Title: ‘**Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way**’ Publication date: **Oct 22, 2019** Publication description: **The 6th IEEE International Conference on Internet of Things: Systems, Management and Security (IOTSMS 2019)**

APPENDICE

APPENDICE A

```
function [ Tdw,Tmigr ] = pre_copy( link_speed,memory_app, Ravg)

step = 1;
tot_sent_pages = 0;
one_page_size = 0.00390625;
tot_page_app = memory_app / one_page_size;
link_speed_by_page = link_speed / one_page_size;
MAX_iteration = 29;
MAX_factor = 3;
MAX_left_page = 50;
Tmigr = 0;
Tdw = 0;
left_pages = tot_page_app;
tot_to_be_sent_pages = left_pages;
IterationFlag = false;
LeftPagesFlag = false;
TotSentPagesFlag = false;


while ((step < MAX_iteration) && (left_pages > MAX_left_page) && ...
       (tot_sent_pages < MAX_factor * tot_page_app ))


    Ti = left_pages/link_speed_by_page;
    Tmigr = Tmigr + Ti;
    tot_sent_pages = tot_to_be_sent_pages;
    avg_dirty_pages = Ravg * Ti;
    one_page_to_be_sent_prob = min(avg_dirty_pages / tot_page_app, 1);
    left_pages = binornd(tot_page_app,one_page_to_be_sent_prob );
    tot_to_be_sent_pages = tot_sent_pages + left_pages;
    if(left_pages > MAX_left_page && ...
      (tot_sent_pages < MAX_factor * tot_page_app ));
        step = step +1;
    end

end

if (~(step < MAX_iteration))
    IterationFlag = true;
    disp(IterationFlag);
end
if (~(left_pages > MAX_left_page))
    LeftPagesFlag = true;
    disp(LeftPagesFlag);
end
if (~(tot_sent_pages < MAX_factor * tot_page_app))
    TotSentPagesFlag = true;
    disp(TotSentPagesFlag);
end
Tdw = left_pages/link_speed_by_page;
Tmigr = Tmigr;
tot_sent_pages = tot_sent_pages + left_pages;

end
```

Nome Variabili	**Significato**
Link_speed	*Velocità collegamento [Mbps]*
Memory_app	*Memoria RAM occupata dall'applicazione [MB]*
R_{avg}	*Average Dirty Rate [page/s]*
T_{dw}	*Service Downtime [s]*
T_{migr}	*Tempo di Migrazione [s]*
MAX_iteration	*29*
MAX_factor	*3*
MAX_left_page	*50*
T_i	*Tempo di migrazione pagina di memoria*
tot_sent_pages	*Totale numero di pagine inviate*
avg_dirty_pages	*Velocità media di sporcamento pagine*
one_page_to_be_sent_prob	*Probabilità di sporcamento di una pagina di memoria*
left_pages	*Pagine sporcate da inviare alla prossima iterazione*
tot_to_be_sent_pages	*Somma delle pagine già inviate e di quelle rimaste*
step	*Passo dell'iterazione*

APPENDICE B

Di seguito il codice sorgente per ricavare le metriche di memoria e progettare il classificatore video.

APPENDICE B1

Il seguente codice di estrazione delle metriche di memoria, inizialmente sviluppato da Brendan Gregg , è stato da me ampliato ai fini della progettazione del classificatore video .

```
use strict;
use Getopt::Long;
use Time::HiRes;
use Text::CSV;

#use Text::CSV_XS;
#use Math::Round;
$| = 1;

sub usage {
    die <<USAGE_END;
USAGE: wss [options] PID duration(s)
    -C          # show cumulative output every duration(s)
    -s secs     # take duration(s) snapshots after secs pauses
    -d secs     # total duration of measurement (for -s or -C)
    -P steps    # profile run (cumulative), from duration(s)
    -t          # show additional timestamp columns
   eg,
    wss 181 0.01       # measure PID 181 WSS for 10 milliseconds
    wss 181 5          # measure PID 181 WSS for 5 seconds (same overhead)
    wss -C 181 5       # show PID 181 growth every 5 seconds
    wss -C -d 10 181 1 # PID 181 growth each second for 10 seconds total
    wss -s 1 181 0.01  # show a 10 ms WSS snapshot every 1 second
    wss -s 0 181 1     # measure WSS every 1 second (not cumulative)
    wss -P 10 181 0.01 # 10 step power-of-2 profile, starting with 0.01s
USAGE_END
}

### options
my $snapshot    = -1;
my $totalsecs   = 999999999;
my $cumulative  = 0;
my $profile     = 0;
my $moretimes   = 0;
my $pausetarget = 0;
GetOptions(
    'snapshot|s=f'  => \$snapshot,
    'duration|d=f'  => \$totalsecs,
    'cumulative|C'  => \$cumulative,
    'profile|P=i'   => \$profile,
    'moretimes|t'   => \$moretimes,
    'pausetarget'   => \$pausetarget,
    'help|h'        => 'usage',
) or usage();
my $pid      = $ARGV[0];
my $duration = $ARGV[1];

if ($pausetarget) {
    print STDERR "--pausetarget disabled (too dangerous). See code.\n";
    exit;

    # if you comment this out, be aware you're sending SIGSTOP/SIGCONTs
    # to the target process, which will pause it, creating latency. If
    # wss.pl crashes or is SIGKILL'd, then the target process can be left
    # in SIGSTOP and will need to be SIGCONT'd manually.
}
if ( @ARGV < 2 || $ARGV[0] eq "-h" || $ARGV[0] eq "--help" ) {
    usage();
    exit;
}
if ( ( !!$cumulative + ( $snapshot != -1 ) + !!$profile ) > 1 ) {
    print STDERR "ERROR: Can't combine -C, -s, and P. Exiting.\n";
    exit;
}
if ( $duration < 0.001 ) {
    print STDERR "ERROR: Duration too short. Exiting.\n";
    exit;
}
my $clear_ref   = "/proc/$pid/clear_refs";
my $smaps       = "/proc/$pid/smaps";
```

```perl
my @profilesecs = ($duration);
my $d;
if ($profile) {
    $d = $duration;
    for ( my $i = 0 ; $i < $profile - 1 ; $i++ ) {
        push( @profilesecs, $d );
        $d *= 2;
    }
}
if ($pausetarget) {
    shift(@profilesecs);
    push( @profilesecs, $d );
}

### main
my ( $rss, $pss, $referenced, $uss, $privateclean, $privatedirty, $shareddirty,
    $dirtypage );
my ( $ts0, $ts1, $ts2, $ts3, $ts4, $ts5 );
my ( $settime, $sleeptime, $readtime, $durtime, $esttime );
my $metric;
my $firstreset = 0;
$sleeptime = 0;
my $fh;

#Create CSV object
my $csv = Text::CSV->new( { sep_char => ";", binary => 1, eol => $/ } )
  or die "Failed to create a CSV handle: $!";
my $filename = "output.csv";
open $fh, ">:encoding(utf8)", $filename or die "failed to create $filename: $!";

### headers
if ($profile) {
    printf
"Watching PID $pid page references grow and dirty pages and number, profile beginning with $duration
seconds, $profile steps...\n";
}
elsif ($cumulative) {
    printf
"Watching PID $pid page references grow and dirty pages and number, output every $duration
seconds...\n";
}
elsif ( $snapshot != -1 ) {
    if ( $snapshot == 0 ) {
        printf
"Watching PID $pid page references , dirty pages and number for every $duration seconds...\n";
    }
    else {
        printf
"Watching PID $pid page references ,dirty pages and number for $duration seconds,repeating after
$snapshot second pauses...\n";
    }
}
else {
    printf
"Watching PID $pid page references and dirty pages and number during $duration seconds...\n";
}
printf "%-7s %-7s ", "Slp(s)", "Dur(s)" if $moretimes;
printf "%-11s %10s %10s %10s %10s %10s %12s %10s %10s\n", "Est(s)", "RSS(MB)",
  "PSS(MB)", "USS(MB)", "Ref(MB)", "PVTDirty(MB)", "SHRDirty(MB)", "Dirty(MB)",
  "#DirtyPageNumber";

$csv->print( $fh, [qw/Sl(s) Dur(s)/] ) if $moretimes;
$csv->print(
    $fh,
    [
        qw/RSS(MB) PSS(MB) USS(MB) WS(MB) PrivDirty(MB) SharedDirty(MB) DirtyPage(MB)
DirtyPage(pages)/
    ]
);

### cleanup
sub cleanup {
    kill -CONT, $pid;
    exit 0;
}
if ($pausetarget) {
    $SIG{INT}  = 'cleanup';    # Ctrl-C
    $SIG{QUIT} = 'cleanup';    # Ctrl-\
```

```
$SIG{TERM} = 'cleanup';     # TERM
}

$ts0 = Time::HiRes::gettimeofday();
while (1) {

    # reset referenced flags
    if ( not $firstreset or $snapshot != -1 or $pausetarget ) {
        kill -STOP, $pid if $pausetarget;
        open CLEAR, ">$clear_ref"
          or die "ERROR: can't open $clear_ref (older kernel?): $!";
        $ts1 = Time::HiRes::gettimeofday();
        print CLEAR "1";
        close CLEAR;
        $ts2        = Time::HiRes::gettimeofday();
        $settime    = $ts2 - $ts1;
        $firstreset = 1;
    }

    # pause
    my $sleep = $duration;
    if ($profile) {
        $sleep = shift @profilesecs;
        last unless defined $sleep;
    }
    kill -CONT, $pid if $pausetarget;
    $ts3 = Time::HiRes::gettimeofday();
    select( undef, undef, undef, $sleep );
    $ts4 = Time::HiRes::gettimeofday();
    kill -STOP, $pid if $pausetarget;

    # read referenced counts
    $rss = $pss = $referenced = $privateclean = $privatedirty = $shareddirty =
      $uss = $dirtypage = 0;
    open SMAPS, $smaps or die "ERROR: can't open $smaps: $!";

    # slurp smaps quickly to minimize unwanted WSS growth during reading:
    my @smaps = <SMAPS>;
    $ts5 = Time::HiRes::gettimeofday();
    close SMAPS;
    kill -CONT, $pid if ( $pausetarget and $snapshot != -1 );
    foreach my $line (@smaps) {
        if ( $line =~ /^Rss:/ ) {
            $metric = \$rss;

        }
        elsif ( $line =~ /^Pss:/ ) {
            $metric = \$pss;

        }
        elsif ( $line =~ /^Referenced:/ ) {
            $metric = \$referenced;
        }
        elsif ( $line =~ /Private_Dirty:/ ) {
            $metric = \$privatedirty;
        }

        elsif ( $line =~ /Private_Clean:/ ) {
            $metric = \$privateclean;
        }

        elsif ( $line =~ /Shared_Dirty:/ ) {
            $metric = \$shareddirty;
        }
        else {
            next;
        }

        # now pay the split cost, after filtering out most lines:
        my ( $junk1, $kbytes, $junk2 ) = split ' ', $line;
        $$metric += $kbytes;
    }

    # uss and dirty_page calculations
    $uss       = $privatedirty + $privateclean;
    $dirtypage = $privatedirty + $shareddirty;
```

```
# time calculations
    if ( $snapshot != -1 or $pausetarget ) {
        $sleeptime = $ts4 - $ts3;
    }
    else {
        $sleeptime += $ts4 - $ts3;
    }
    $readtime = $ts5 - $ts4;
    $durtime  = $ts5 - $ts1;
    if ($pausetarget) {
        $esttime = $ts4 - $ts3;
    }
    else {
        $esttime = $durtime - ( $settime / 2 ) - ( $readtime / 2 );
    }

    # output
    printf "%-15.3f %-7.3f ", $sleeptime, $durtime if $moretimes;
    printf
      "%-15.3f %-10.3f %-10.2f %-10.2f %-10.2f %-10.2f %-10.2f %10.2f %10.2f\n",
      $esttime,
      $rss / 1024, $pss / 1024, $uss / 1024, $referenced / 1024,
      $privatedirty / 1024, $shareddirty / 1024, $dirtypage / 1024,
      $dirtypage / 4;

    # output csv
    $csv->print( $fh, [ $sleeptime, $durtime ] ) if $moretimes;

    #CosÃ¬ riesco a salvare con tre cifre decimali
    $csv->print(
        $fh,
        [
            int( ( $rss / 1024 ) * 1000 ) / 1000,
            int( ( $pss / 1024 ) * 1000 ) / 1000,
            int( ( $uss / 1024 ) * 1000 ) / 1000,
            int( ( $referenced / 1024 ) * 1000 ) / 1000,
            int( ( $privatedirty / 1024 ) * 1000 ) / 1000,
            int( ( $shareddirty / 1024 ) * 1000 ) / 1000,
            int( ( $dirtypage / 1024 ) * 1000 ) / 1000,
            $dirtypage / 4
        ]
    );

    # snopshot sleeps
    if ( $snapshot != -1 ) {
        select( undef, undef, undef, $snapshot );
    }
    elsif ( not $cumulative and not $profile ) {
        last;
    }

    if ( $ts5 - $ts0 >= $totalsecs ) {
        last;
    }
}
close $fh or die "failed to close my output.csv: $!";
kill -CONT, $pid if $pausetarget;
```

APPENDICE B2

Il seguente codice è stato utilizzato per determinare il migliore layout di rete MLP per il nostro classificatore video.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report

if __name__ == '__main__':
    np.set_printoptions(threshold=np.inf)
    pathDataSet = 'D:\\vmshare\\Risultati\\dataset.csv'
    pathTargetLabel = 'D:\\vmshare\\Risultati\\target_label.csv'
    target_names = ['SD', 'HD_720', 'HD_1080','UHD']
    dataset = np.genfromtxt(fname=pathDataSet,delimiter=';')
    targetLabel = np.genfromtxt(fname=pathTargetLabel,delimiter=';',dtype='int')
    le = LabelEncoder()
    targetLabelEncoded = le.fit_transform(targetLabel)
    targetLabelEncoded2D = targetLabelEncoded.reshape(-1,1)
    ohe = OneHotEncoder(categorical_features = [0],dtype = np.int)
    targetLabelOneHot = ohe.fit_transform(targetLabelEncoded2D).toarray()
    X_train, X_test, Y_train, Y_test =
train_test_split(dataset,targetLabelEncoded,test_size=0.3,random_state=42)
    scaler = StandardScaler().fit(X_train)
    X_train_transformed = scaler.transform(X_train)
    X_test_transformed = scaler.transform(X_test)
    pipeline =
make_pipeline(StandardScaler(),MLPClassifier(hidden_layer_sizes=(3,),activation="logistic
",solver='lbgfs',alpha=1e-01,random_state=42,tol=1e-4,max_iter=10000))
    hidden_layer_size_range = [3,4,5,6,7,8,]
    activation_range = ['logistic','tanh','relu']
    solver_range = ['lbfgs','sgd']
    alpha_range = [1e-1,1e-2,1e-3,1e-4,1e-5]
    learning_rate_range = ['constant','adaptive']
    param_grid = {'mlpclassifier__hidden_layer_sizes': hidden_layer_size_range,
'mlpclassifier__activation': activation_range,'mlpclassifier__solver':
solver_range,'mlpclassifier__alpha': alpha_range,'mlpclassifier__learning_rate':
learning_rate_range}
    gs = GridSearchCV(pipeline,param_grid,scoring='accuracy',n_jobs=-1,cv=5)
    gs = gs.fit(X_train,Y_train)
    print("The best value in terms of accuracy is : " , gs.best_score_)
    print()
    print("The best parameters set found on development set : ",gs.best_params_)
    print()
    means_test_score = gs.cv_results_['mean_test_score']
    stds_test_score = gs.cv_results_['std_test_score']
    rank_testScore = gs.cv_results_['rank_test_score']
    print("Rank, Mean, Standard deviation score and related grid on development set")
    print()
    for rank, mean, std, params in
zip(rank_testScore,means_test_score,stds_test_score,gs.cv_results_['params']):
        print("%i  %0.5f (+/-%0.5f) for %r" %(rank,mean,2*std,params))
    print()
    print("Classification Report\n")
    print("The model is trained on the full development set.")
    print("The scores are computed on the full evaluation set.")
    print()
    Y_true, Y_pred = Y_test, gs.predict(X_test)
    print(classification_report(Y_true,Y_pred,target_names=target_names))
    clf = gs.best_estimator_
    clf.fit(X_train, Y_train)
    print('Test accuracy: %.3f' % clf.score(X_test, Y_test))
```

APPENDICE C

Quello che segue è solamente la parte centrale per il calcolo del percorso migliore.

```
while ((step < MAX_iteration) && (left_pages > MAX_left_page) && (tot_sent_pages <
MAX_factor * tot_page_app ))
    %path1 %Calcolo Ti P1
    Ti_p1 = left_pages/vet_bw_disp_p1_by_page(counter_bw_disp_px);
    %path2 %Calcolo Ti P2_L1
    Ti_p2_l1 = left_pages/vet_bw_disp_p2_l1_by_page(counter_bw_disp_px);
    %Calcolo Ti P2_L2
    Ti_p2_l2 = left_pages/vet_bw_disp_p2_l2_by_page(counter_bw_disp_px);
    %chiamata al controller
    Ti_px = Controller(Ti_p1,Ti_p2_l1,Ti_p2_l2);
    %vettore dei Ti
    Ti_vet_px(step) = Ti_px;
    %Update Migration time
    Tmigr_px = Tmigr_px + Ti_px;
    %Update total sent pages
    tot_sent_pages = tot_to_be_sent_pages;
    %ith average dirty pages
    avg_dirty_pages = Ravg * Ti_px;
    %One dirty page probability according to binomial distribution
    one_page_to_be_sent_prob = min(avg_dirty_pages / tot_page_app, 1);
%Update left pages, total pages to be sent and step %binornd(N,p) restituisce un
numero tra 0 e N secondo una distribuzione binomiale
    left_pages = binornd(tot_page_app,one_page_to_be_sent_prob );
%Total pages to be sent next iteration if  tot_sent_pages < MAX_factor *
tot_page_app
    tot_to_be_sent_pages = tot_sent_pages + left_pages;
if(left_pages > MAX_left_page && (tot_sent_pages < MAX_factor * tot_page_app ))
    step = step +1;
    counter_bw_disp_px = counter_bw_disp_px + 1;
    end
end
```

BIBLIOGRAFIA

1. ***[Online] https://spectrummattersindeed.blogspot.com/2019/05/how-much-will-5g-data-usage-increase.html.***

2. **[Online] "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015-2020".** ***2016.***

3. **”Emerging technologies and research challenges for 5G wireless networks”.** ***W. Chin, Z. Fan, R. Haines. s.l. : IEEE Wireless Commun, vol. 21, no. 2, Apr. 2014.***

4. **”Leveraging SDN for The 5G Networks: Trends, Prospects and Challenges”.** ***Akram Hakiri, Pascal Berthou. June 2015.***

5. **"Access, Fronthaul and Backhaul Networks for 5G & Beyond".** ***Muhammad Ali Imran, Syed Ali Raza Zaidi , Muhammad Zeeshan Shakir. Stevenage : Institution of Engineering and Technology, 10 Oct 2017.***

6. **"Review of latest advances in 3GPP standardization: D2D communication in 5G systems and its energy consumption models".** ***M. Höyhtyä, O. Apilo , M. Lasanen. 2018.***

7. **"Full dimension MIMO for LTE-Advanced and 5G".** ***Young-Han Nam, Md Saifur Li Rahman, Yang Li, Gang Xu, Eko Onggosanusi, Jianzhong Zhang, Ji-Yun Seol. 2015.***

8. **[Online] https://developer.samsung.com/tech-insights/5G/5g-key-enabling-technologies.**

9. A. Silberschatz, P. B. Galvin and G. Gagne. **Operating System Concepts.** ***s.l. : Wiley, 2013.***

10. **[Online] https://www.intel.it/content/www/it/it/virtualization/virtualization-technology/intel-virtualization-technology.html.**

11. **[Online] https://www.amd.com/en/technologies/virtualization.**

12. **[Online] https://www.linux-kvm.org/page/Main_Page.**

13. **[Online] https://www.virtualbox.org/.**

14. **[Online] https://www.docker.com/.**

15. **[Online] https://www.opencontainers.org/.**

16. **[Online] https://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html.**

17. **"The working set model for program behavior".** ***Peter J. Denning. 1967. Proceedings of the ACM symposium on Operating System Principles.***

18. **[Online] https://en.wikipedia.org/wiki/Generalized extreme value distribution.**

19. **[Online] https://www.opennetworking.org/m-cord/.**

20. **[Online] https://www.opendaylight.org/.**

21. **[Online] https://www.openstack.org/.**

22. **[Online] https://kubernetes.io/.**

23. **[Online] https://wiki.opencord.org/display/CORD/CORD+Reference+Implementation.**

24. **[Online] https://it.wikipedia.org/wiki/OpenFlow.**

25. **[Online] https://www.docker.com/why-docker.**

26. **[Online] https://onosproject.org/.**

27. **[Online] https://guide.xosproject.org/.**

28. **[Online] https://www.ansible.com/.**

29. **[Online] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.**

30. **[Online] http://openvirtex.com/.**

31. **"Live Migration of Virtual Machines Based on DPDT".** ***Danwei Chen, Hanbing Yang, Qinghan Xue, Yong Zhou. Springer, Berlin, Heidelberg, 2013, Communications in Computer and Information Science, Vol. 334.***

32. **[Online] https://xenproject.org/.**

33. **"A time-series based precopy approach for live migration of virtual machines".** ***Li Bolin, Hu Zhou, Lei Yu, Lei Dong, Xu Jiandun. Tainan : s.n., 2011.***

34. **"Live Service Migration in Mobile Edge Clouds".** ***A. Machen, S. Wang, Kin K. Leung, Bong Jung Ko, Theodoros Salonidis. Vol. 99, pp. 2-9.***

35. Valencia Conference. [Online] http://emergingtechnet.org/MCSMS2018/.

36. **"Adaptive task allocation for mobile edge learning".** ***Sorour, Umair Yaqub and Sameh. Nov. 2018.***

37. **[Online] https://www.cisco.com/c/dam/m/en_in/innovation/enterprise/assets/mobile-white-paper-c11-520862.pdf.**

38. **"On-the-fly QoE-aware transcoding in the mobile edge".** ***S. Dutta, T. Taleb, P. A. Frangoudis, and A. Ksentini. 2016. IEEE Global Communications Conference.***

39. **"NFV-Based Mobile Edge Computing for Lowering Latency of 4K Video Streaming".** ***Linh Van Ma, Van Quan Nguyen, Jaehyung Park, Jinsul Kim. 2018. International Conference on Ubiquitous and Future Networks.***

40. **"Collaborative multi-bitrate video caching and processing in Mobile-Edge Computing networks".** ***Tuyen X. Tran, P. Pandey, A. Hajisami, D. Pompili. 2017. 13th Annual Conference on Wireless On-Demand Network Systems and Services (WONS).***

41. **"Video rate adaptation and traffic engineering in mobile edge computing and caching-enabled wireless networks".** ***C. Liang, Y. He, F. Richard Yu, and N. Zhao. 2018. IEEE Vehicular Technology Conference.***

42. **[Online] https://github.com/brendangregg/wss.**

43. **[Online] https://en.wikipedia.org/wiki/Random_forest.**

44. **[Online] https://en.wikipedia.org/wiki/Multilayer_perceptron.**

45. **[Online] https://scikit-learn.org/stable/.**

46. **[Online] https://github.com/rasbt/mlxtend.**

47. Swingler. 1996, p. 53.

48. Linoff, Berry and. 1997, p. 323.

49. Blum. 1992, p. 60.

50. **"Approximation capabilities of multilayer feedforward networks Neural Networks".** *Kurt Hornik. 1991.*

51. **[Online] http://www.ict-tropic.eu/.** *2012.*

52. **"Mobile micro-cloud: Application classification, mapping, and deployment".** *Shiqiang Wang, Guan-Hua Tu, Raghu K. Ganti, Ting He, Kin Leung, Howard Tripp, K Warr, Murtaza Zafer. New York : s.n., 2013. Proc. Annu. Fall Meeting ITA (AMITA).*

53. **"MobiScud: A fast moving personal cloud in the mobile network".** *Kaiqiang Wang, Minwei Shen, Junguk Cho, Arijit Banerjee, Jacobus van der Merwe, Kirk Webb. London : s.n., 2015. Proc. Workshop All Things Cellular Oper. Appl. Challenge.*

54. **"Follow-Me Cloud: When Cloud Services Follow Mobile Users".** *Tarik Taleb, Adlen Ksentini, Pantelis A. Frangoudis.*

55. **"CONCERT: A cloudbased architecture for next-generation cellular systems".** *Jingchu Liu, Tao Zhao, Sheng Zhou, Yu Cheng, and Zhisheng Niu. 2014. Vol. 21, p. 1422.*

56. **"Fast transparent migration for virtual machines".** *Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Anaheim, CA, USA : s.n., 2005. USENIX Annu. Tech. Conf. Gen. Track.*

57. **"Improving the live migration process of large enterprise applications".** *Stuart Hacking, Benoît Hudzia. Barcelona, Spain : s.n., 2009. 3rd Int. Workshop Virtual. Technol. Distrib. Comput.*

58. **"Aspects of cache memory and instruction buffer performance".** *Hill Mark Donald. Berkeley, CA, USA : s.n., 1987.*

59. **"Efficient wide-area live virtual machine migration using distributed content-based addressing".** *Pierre Riteau, Christine Morin, and Thierry Priol. 2010.*

60. **"CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines".** *Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, Jacobus Van der Merwe, Jinho Hwang, Guyue Liu, Lucas Chaufournier. 2011. Vol. 46, p. 121132.*

61. **"High performance virtual machine migration with RDMA over modern interconnects".** *Wei Huang, Qi Gao, Jiuxing Liu, Dhabaleswar K. Panda. Austin, TX, USA : s.n., 2007. Int. Conf. Cluster Comput.*

62. **"InfiniBand Architecture Specification: Release 1.0".** *Portland, OR, USA, 2000 : InfiniBand Trade, 2000.*

63. **"Live migration of virtual machine based on full system trace and replay".** *Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. 2009. 18th ACM Int. Symp. High Perform. Distrib. Comput.*

64. **"High performance live migration through dynamic page transfer reordering and compression".** ***Petter Svard, Johan Tordsson, Benoit Hudzia, Erik Elmroth. Athens, Greece : s.n., 2011. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci. (CloudCom).***

65. **"Optimizing the live migration of virtual machine by CPU scheduling".** ***Hai Jin, Wei Gao, Song Wu, Xuanhua Shi, Xiaoxin Wu, Fan Zhou. 2011. Vol. 34, p. 10881096.***

66. **"How fast can you reconfigure your partially".** ***Sieber, Christian, Durner, Raphael and Kellerer, Wolfgang. Stockholm, Sweden : s.n., 12-16 June 2017.***

67. **"Network Link Dimensioning based on Statistical Analysis and Modeling of Real Internet Traffic".** ***Mohammed Alasmar, Nickolay Zakhleniuk. 2017.***

68. **[Online] https://it.mathworks.com/help/stats/gevrnd.html.**

69. **[Online] https://www.cisco.com/c/en/us/solutions/collateral/serviceprovider/global-cloud-index-gci/white-paper-c11-738085.html.**

70. **[Online] https://www.docker.com/why-docker.**

71. **[Online] https://diamanti.com/wp-content/uploads/2018/07/WP Diamanti EndUser Survey 072818.pdf.**

72. **[Online] http://cve.mitre.org/cgibin/cvename.cgi?name=CVE-2019-5736.**

73. **[Online] https://linuxcontainers.org/.**

74. **[Online] http://mesos.apache.org/.**

75. **"Efficient Live Migration of Edge Services Leveraging Container Layered Storage".** ***Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. IEEE Transactions on Mobile Computing, 2018.***

76. **[Online] https://en.wikipedia.org/wiki/Defense in depth (computing) .**

77. **[Online] https://katacontainers.io/.**

78. **[Online] https://grpc.io/.**

79. **[Online] https://github.com/hashicorp/yamux.**

80. **[Online] https://github.com/katacontainers/qemu/tree/qemu-lite-2.11.0.**

81. **[Online] https://en.wikipedia.org/wiki/NVDIMM.**

82. **[Online] https://en.wikipedia.org/wiki/Singleroot input/output virtualization.**

83. **[Online] Available https://wiki.osdev.org/Virtio.**

84. **[Online] https://virt.kernelnewbies.org/MacVTap.**

85. **[Online] https://www.kernel.org/doc/Documentation/filesystems/9p.txt.**

86. **[Online] https://en.wikipedia.org/wiki/Device mapper.**

87. **[Online] https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.**

88. **[Online] https://wiki.archlinux.org/index.php/AppArmor.**

89. **[Online] https://wiki.archlinux.org/index.php/SELinux.**

90. **[Online] https://wiki.opencord.org/pages/viewpage.action?pageId=1278081.**

91. **"XOS: An Extensible Cloud Operating System".** ***Larry Peterson, Scott Baker, Marc De Leenheer, Andy Bavier. 2015.***

92. **[Online] https://www.cisco.com/c/dam/m/en_in/innovation/enterprise/assets/mobile-white-paper-c11-520862.pdf.**

93. **"On-the-fly QoE-aware transcoding in the mobile edge".** ***Sunny Dutta, Tarik Taleb, Pantelis A. Frangoudis, and Adlen Ksentini. 2016. IEEE Global Communications Conference, GLOBECOM 2016.***

94. **"NFV-Based Mobile Edge Computing for Lowering Latency of 4K Video Streaming".** ***Linh Van Ma, Van Quan Nguyen, Jaehyung Park, Jinsul Kim. 2018. International Conference on Ubiquitous and Future Networks (ICUFN).***

95. **"Evaluation of delta compression techniques for efficient live migration of large virtual machines".** ***Petter Svärd, Benoit Hudzia, Johan Tordsson, Erik Elmroth. 2011. Vol. 46, p. 111120.***