POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Orchestrazione di servizi virtualizzati su infrastruttura Fog/Edge/Cloud computing per applicazioni Smart Grid in sistemi elettro-energetici di potenza.

Politecnico
di Torino

Relatore
Prof. Fulvio Risso

Candidato
Sergio Di Pane
matr. 177289

Anno Accademico 2020/2021

Indice dei contenuti

1 Abstract

With renewable and alternative energy power sources available today, such as wind or solar, the share of decentralized power generation is increasing, which causes general changes and challenges for the conventional power system. Hence, transmission and distribution power operators need new and emerging technologies to ensure the reliable control, efficiency, stability and security of the entire electrical power system, which can be achieved by a massive use of ICT technologies.

As a feasible solution for structuring the forthcoming intelligent power grids, the Fog/Edge/Cloud (FEC) computing hierarchical paradigm has been introduced by the computing ICT world, providing an innovative and reasonable way of reliable, scalable and secure implementation for future-oriented power systems called Smart Grids. In fact, SGs are thought as the next generation power grids in which the electricity management and distribution processes are digitally designed to control, monitor, protect and real-time automatically and adaptively optimize bidirectional flows both of energy and information between utility providers and consumers, thus allowing not only the integration of green power resources into the energy distribution system but, most of all, the dynamic balance between energy supply and power usage demand.

This thesis work provides an initial overview of the FEC computing approach when applied to the specific domain of electrical power grid, focusing particularly on Fog/Edge platforms, which are based on virtualization and orchestration technologies. The main idea is to move computing resources at the edge of the network, close to where data is generated and consumed. Thanks to their characteristics in terms of agility, low communication latencies, lower bandwidth consumption and better security, they could lead to the development and implementation of various next-generation applications and services, their quick (or automated) deployment, addition and update in a cost-effective way, regardless of the presence of dedicated physical hardware.

The above features of Fog/Edge computing could be relevant to the power grid environment, characterized by massively distributed hardware / software architecture, widespread geographical distribution and very high security and resilience requirements. This research activity aims at the preliminary design of the Fog/Edge/Cloud Computing technologies in the power grid environment, assessing their advantages (and possible drawbacks) compared to the “traditional” approach based on dedicated hardware appliances. The presented technologies will be evaluated by analyzing some software services that are particularly significant for the power grid environment, including the assessment of some key performance indicators in real use cases, hence providing a preliminary indication about the appropriateness of the above technologies while adopted in the target Smart Grid scenario.

2 Introduzione

Le piattaforme Fog/Edge computing possono introdurre le caratteristiche di agilità cui siamo abituati nel mondo Cloud anche nelle porzioni periferiche dell'infrastruttura IT. Rispetto al tradizionale paradigma Cloud, l'Edge computing sposta le risorse di computing ai margini della rete, vicino al luogo in cui i dati vengono usualmente generati e consumati, introducendo così una serie di vantaggi quali basse latenze, minor consumo di banda, maggiore sicurezza. Tali vantaggi consentono lo sviluppo e il deployment di applicazioni di nuova generazione, aprendo le porte a nuove possibilità per i grandi e piccoli fornitori di servizi.

Nascendo come estensione del cloud, le piattaforme Fog/Edge possono essere particolarmente appropriate per il deployment rapido (o automatizzato) di applicazioni e servizi, consentendo la loro aggiunta e aggiornamento indipendentemente dalla presenza di hardware fisico dedicato grazie alle caratteristiche di trasparenza delle applicazioni virtualizzate (o *cloud-native*). In aggiunta, possono consentire l'applicazione di criteri di sicurezza particolarmente aggressivi, quali ad esempio la protezione dell'infrastruttura attraverso disconnessione dalla rete di backbone, con lo spostamento (temporaneo) dei servizi software essenziali in locale e il successivo riallineamento dei dati con l'istanza "master" remota quando se ne ripresenti il momento opportuno.

Tali caratteristiche possono rivelarsi particolarmente importanti per l'ambiente elettro-energetico, caratterizzato da (a) una architettura hardware/software massimamente distribuita, (b) una distribuzione geografica capillare, (c) requisiti di sicurezza e resilienza molto alti.

L'attività di ricerca condotta nell'ambito di questa tesi si è proposta di progettare la migrazione dei **processi di misura e controllo** dell'attuale power grid verso tecnologie **Fog/Edge/Cloud Computing** (**Smart Grid** – SG), esaminando alcuni componenti funzionali e servizi software particolarmente significativi per l'esecuzione e il monitoraggio delle prestazioni di casi d'uso reali in ambiente elettro-energetico. L'obiettivo di questo lavoro è costituire una base concettuale, tecnologica e implementativa utile agli sviluppi delle Smart Grid, le reti elettriche di nuova generazione in cui i processi di gestione e distribuzione dell'elettricità sono progettati incorporando flussi di energia e di informazioni bidirezionali tra fornitori e consumatori, integrando tecnologie digitali avanzate con capacità di elaborazione e pervasività tali da migliorare controllo, efficienza, affidabilità e sicurezza della rete stessa. Una Smart Grid interagisce con gli apparati intelligenti dei consumatori on-premise, utilizzando pienamente le flessibilità disponibili e conseguire obiettivi di risparmio ed efficienza energetica complessiva del sistema, di riduzione dei costi e aumento dell'affidabilità in un contesto caratterizzato dalla crescente diffusione di fonti energetiche rinnovabili non programmabili. Essa è una modernizzazione della rete elettrica originaria, soprattutto a livello di distribuzione dell'energia e di gestione delle infrastrutture elettriche attraverso l'integrazione delle moderne tecnologie di comunicazione per scambiare informazioni in tempo reale tra utilizzatori, gestori e fornitori di energia.

Questa tesi descrive dello stato di avanzamento del progetto, ed è suddivisa come segue.

La Sezione 3 analizza e discute le tecnologie e le soluzioni ICT abilitanti per gli scenari Fog/Edge, con valutazioni comparative dei vari approcci disponibili.

La Sezione 4 descrive e analizza i due casi applicativi reali che questo progetto prende come riferimento: *(i)* osservabilità della rete tramite acquisizione di misure da *Phasor Measurement Units* (PMUs), e *(ii)* controllo di una Microrete di distribuzione elettrica (potenzialmente isolata). L'applicabilità dei paradigmi Fog/Edge agli scenari applicativi di riferimento, prendendo in considerazione i singoli componenti coinvolti, relativi KPI, e individuando vantaggi e svantaggi delle possibili configurazioni rispetto a una serie di eventi ICT critici, viene invece studiata nella Sezione 5.

La Sezione 6 presenta la soluzione architetturale proposta in questo progetto alla luce delle considerazioni sviscerate nelle due sezioni precedenti. Nello specifico, si è delineata l'attività di orchestrazione dei componenti funzionali virtualizzati della Smart Grid in ambiente mono-cluster.

Nella Sezione 7 viene presentato lo stato di avanzamento dell'attività, evidenziando i problemi tecnici riscontrati e dettagliando le soluzioni adottate.

Infine, la Sezione 8 delinea un piano relativo ai prossimi step da effettuare per proseguire efficacemente l'attività di ricerca futura avviata e tracciata con questo lavoro.

3 Stato dell'arte

Questa sezione presenta i concetti di *virtualizzazione* e *orchestrazione*, analizzando i dettagli delle tecnologie e soluzioni attualmente disponibili, quali i diversi tipi di virtualizzazione, le particolarità della "Operating-System Level Virtualization", meglio conosciuta come *containerizzazione*, e le principali architetture di orchestrazione, sia per ambienti single-cluster che multi-cluster. Per ogni tecnologia e concetto presentato verranno effettuati confronti qualitativi e discussi vantaggi e svantaggi.

3.1 Virtualizzazione

La virtualizzazione consiste nel sostituire hardware dedicati a specifiche funzionalità con hardware di computing generico, sul quale vengono allocate delle risorse "virtualizzate" ed eseguiti i singoli servizi sotto forma di pacchetti funzionali facilmente distribuibili (es., macchine virtuali – VM, containers).

Le tecnologie di virtualizzazione oggi disponibili sul mercato sono parecchio eterogenee e sfruttano differenti paradigmi. Nella *Full Software Virtualization*, tutto l'hardware viene emulato all'interno dell'esecuzione di un software, sul quale poi viene eseguito un SO completo, che crede di agire su una infrastruttura fisica. La *Hardware-Assisted Software Virtualization* differisce dalla precedente per il fatto che alcune funzionalità hardware vengono "accelerate", grazie alla "collaborazione" dell'hardware sottostante, che fornisce un supporto nativo alla virtualizzazione. Un approccio differente è la *Paravirtualization*, dove l'hardware non viene virtualizzato al 100%, per cui il guest OS agirà in maniera consapevole interfacciandosi in parte con l'hardware sottostante. Questo permette un minore overhead (l'hardware non è completamente replicato in software) e la possibilità di interfacciarsi direttamente con i driver dell'host OS, riducendo anche l'overhead di esecuzione.

3.1.1 Macchine virtuali e containers

La containerizzazione è anche conosciuta come *"Operating-system level virtualization"*. In questo particolare tipo di "virtualizzazione" il kernel del sistema operativo in esecuzione sul device host permette l'esistenza di più istanze in user-space isolate tra loro, dette "containers". Dal punto di vista delle applicazioni in esecuzione all'interno di ciascun container, questo appare come una macchina a sé stante, da cui sono visibili soltanto le risorse, i dispositivi, i file, e i dispositivi hardware assegnati a quel container.

Il vantaggio principale della containerizzazione rispetto alle tecniche di virtualizzazione più tradizionali consiste in una massiccia riduzione dell'overhead: le applicazioni in esecuzione all'interno di un container utilizzano le system call del kernel host, senza dunque il bisogno dello

strato di emulazione aggiuntivo delle macchine virtuali. Inoltre, non è necessario che l'hardware sia ottimizzato (es., hardware-assisted software virtualization) per ottenere performance efficienti. Un ulteriore vantaggio è la velocità di creazione, esecuzione e cancellazione dei containers.

Viceversa, il principale svantaggio della containerizzazione rispetto alla virtualizzazione riguarda la minor flessibilità: mentre su un VM hypervisor è possibile eseguire qualsiasi sistema operativo (es., una VM Windows può essere lanciata su un hypervisor Linux), il sistema operativo di tutti i container deve coincidere con quello in esecuzione sulla macchina host, dato che il kernel è condiviso. Inoltre, a seguito della condivisione del kernel Linux tra i vari containers, un problema di sicurezza del kernel può ripercuotersi a catena su tutti i containers in esecuzione sul sistema host. Il problema è meno sensibile nel caso delle VM in quanto (1) la VM definisce una interfaccia non banale da superare anche nel caso in cui l'hypervisor sia compromesso; (2) un hypervisor è tipicamente più piccolo rispetto al kernel di un OS tradizionale, pertanto offre una minor superficie di attacco.

Esistono varie soluzioni che implementano il concetto di containerizzazione. LXC [LXC] è sfruttato da varie soluzioni di containerizzazione: combina i *cgroups* del kernel Linux e il supporto a differenti *namespace* per implementare gli ambienti d'esecuzione isolati. Con LXC è possibile supportare molte delle funzionalità di isolamento tipiche dei containers (file system, CPU quotas, networking) ma alcune funzionalità, come disk quotas e I/O limits, sono supportate soltanto parzialmente. Un engine che supporta la creazione di containers Windows è invece Hyper-V Containers [Thurrott2007]. Docker [Barbier2014], che nelle sue prime versioni si basava su LXC, è oggi la soluzione più popolare di containerizzazione, che fornisce numerose funzionalità avanzate e tool accessori di utilità come Docker Compose [DockerDocs2017a], che permette di definire e lanciare applicazioni costituite da più containers, e Docker Swarm [DockerDocs2017b], che fornisce clustering nativo tra Docker engines differenti. In aggiunta, Docker è il sistema di containerizzazione adottato dai principali software di orchestrazione (es., Kubernetes)[1].

3.1.1.1 Docker

Allo stato corrente le scelte implementative tendono verso l'utilizzo di Docker. Questa tecnologia, che nelle sue prime versioni si basava su LXC, è oggi la piattaforma software più popolare di containerizzazione che permette di creare, testare e distribuire applicazioni con la massima rapidità creando un *lightweight*, *portable* e *self-contained* package. Docker raccoglie il software in unità standardizzate chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime che, indipendentemente dall'ambiente di

[1] Recentemente (dicembre 2020) il progetto Kubernetes ha annunciato l'abbandono del runtime Docker in Kubernetes, sostituito da *containerd*. Tuttavia questo è un cambiamento che non impatta i container Docker attualmente creati, che possono effettivamente essere eseguiti sia con il runtime Docker originale che con containerd.

esecuzione, può essere eseguito ovunque. L'applicazione containerizzata dunque assume sempre lo stesso comportamento di esecuzione su qualsiasi host senza essere vincolata alle caratteristiche di sistema quali versione kernel e/o distribuzione nella quale viene lanciata.

Di seguito sono elencate le principali features aggiuntive che Docker offre rispetto alla soluzione LXC e che risultano rilevanti per l'implementazione descritta in questa sezione.

- Application portability: come detto in precedenza, Docker permette di creare/distribuire ed eseguire self contained package, incapsulante l'applicazione da eseguire, su qualsiasi dispositivo di computing in maniera indipendente dall'ambiente di esecuzione;
- Union File System (*UFS*): considerato che i container replicano l'intero file system necessario per la corretta esecuzione, l'image-size del container stesso può essere elevata (es., 100 MB o più), dunque l'avvio del container potrebbe causare un ritardo non trascurabile (download time di 100 MB è circa 1 s in una 1 Gbps network). Tramite una visione stratificata del file system, Union File System (es., overlayFS) permette a più container di condividere tra loro il loro base-file system (se già presente sul dispositivo di computing) evitando così di scaricare l'intera immagine; procedendo, solamente i file non presenti verranno scaricati e andranno ad affiancare, al top, lo shared-file system differenziando l'ambiente di un container da un altro e riducendo i tempi di download in maniera notevole.
- Automatic build: Docker permette, in maniera automatica, di creare un container personalizzato partendo dalla specifica dei suoi componenti base (es. specifica del SO, download di determinate librerie) e delle azioni da eseguire, tramite la stesura di un *Dockerfile*. Esso è uno YAML file assimilabile ad una ricetta ben strutturata, composta da più steps scritti dal developer che, se condotti bene e in maniera sequenziale nel tempo, creano il container desiderato.
- Integration con Kubernetes: Docker è il sistema di containerizzazione adottato dai principali software di orchestrazione (es., Kubernetes) usato per creare ed eseguire le applicazioni desiderate in container logicamente incapsulati in Pod.
- Isolation: come avviene anche per LXC, l'isolamento di un Docker container con eventuali altri Docker container dispiegati nell'intero cluster è una feature consolidata e funzionante (anche se deployati sullo stesso nodo di computing).
- Boot speed: tramite Docker engine, la procedura di pull (download) dell'immagine di un Docker container da un repository (es., DockerHub) e l'esecuzione stessa è semplice e veloce.

3.1.2 Vantaggi e svantaggi delle tecnologie di virtualizzazione

I vantaggi e gli svantaggi della virtualizzazione sono schematizzati nella tabella seguente.

Vantaggi	Svantaggi
Isolamento	**Overhead**
✓ Servizi diversi possono essere eseguiti sullo stesso host fisico in ambienti isolati (VM, containers).	× L'esecuzione della stessa applicazione in modalità virtuale può richiedere maggiori risorse hardware

- ✓ Crash o vulnerabilità di sicurezza in un servizio rimangono isolate all'interno della VM, non intaccando gli altri servizi.
- ✓ OS differenti possono essere utilizzati in differenti VM.

Consolidamento

- ✓ OS differenti possono essere eseguiti contemporaneamente sulla stessa macchina fisica.
- ✓ Ottimizza l'utilizzo delle risorse delle moderne architetture CPU multi-core.
- ✓ Migliora i consumi energetici, evitando la presenza di macchine attive ma quasi totalmente sottoutilizzate (c.d. *idle server*); infatti il consumo energetico è poco influenzato dal carico computazionale sulla macchina stessa.
- ✓ Gestione semplificata dell'hardware in datacenter centralizzati.
- ✓ CAPEX non proibitivi sul requisito della high availability (HA), con possibilità di acquistare risorse ridondate organizzate in clustering.

Flessibilità e agilità

- ✓ Controllo completo sulle informazioni di esecuzione delle VM/containers, con possibilità di sospendere/riprendere l'esecuzione del guest OS / OS virtuale (anche su server diversi).
- ✓ Live Migration di VM/container su hardware differenti, ad esempio per motivi di manutenzione, guasti od ottimizzazione del servizio.
- ✓ Scale-up: le risorse allocate a una certa VM/container possono essere dinamicamente ridimensionate in base al carico.
- ✓ Scale-out: le VM/container possono essere replicate (su stesso hardware o su più macchine) per far fronte a picchi sul carico di lavoro.
- ✓ È possibile istanziare nuovi servizi al volo, lanciando le opportune VM/containers.

Enabler

- ✓ Abilita Fog/Edge/Cloud Computing.
- ✓ Data la loro flessibilità, è possibile automatizzare il deployment e la configurazione dei servizi virtuali.

rispetto all'esecuzione *bare-metal* su dispositivi fisici, particolarmente per le VM; l'overhead si esecuzione è invece ridotto nel caso dei containers.

- × Ogni applicazione può essere sviluppata per un certo OS, quindi vincolata ad eseguire su tale OS, per cui è necessario gestire lo strato di software extra (es. hypervisor) necessario alla virtualizzazione VM; questo problema è meno determinante nei containers.

Difficoltà di gestione

- × La complessità dell'architettura aumenta a causa del maggiore numero di componenti coinvolti (es. hypervisor).
- × Maggiore difficoltà nel supportare hardware eterogeneo.
- × Difficoltà nell'uniformare l'accesso a componenti speciali (es., GPU).
- × È opportuno accoppiare le tecnologie di virtualizzazione con un sistema di **orchestrazione**, in quanto l'allocazione delle risorse fisiche non è adattiva e, in mancanza di un orchestratore, va effettuata manualmente.

3.2 Orchestrazione

Per orchestrazione si intende *l'automazione dei processi di deployment, gestione e configurazione dei servizi virtuali*. Si tratta di una metodologia altamente funzionale per i contesti enterprise che mira a distribuire, scalare e gestire i carichi di lavoro (*workloads*) di centinaia o migliaia di host (c.d. *computing nodes*, nodi di computing di orchestrazione) le cui risorse sono sfruttate per eseguire containers (*"Operating-system level virtualization"*) / VMs (*"Full-Virtualization"*) secondo le accezioni viste in precedenza. L'insieme di compute nodes costituisce l'infrastruttura di orchestrazione organizzata in *cluster* (ad es., l'insieme dei server in un datacenter). Esistono differenti soluzioni per l'orchestrazione di servizi virtuali, quali soluzioni di orchestrazione basate su macchine virtuali come OpenStack [Sefraoui2021], Open Source Mano [ETSI2016], Apache CloudStack [Sabharwal2013], e soluzioni basate sui containers, come Red Hat OpenShift [Pousty2014] e Kubernetes [Burns2016].

È un dato di fatto che le tecnologie di orchestrazione dei containers, principalmente Kubernetes, stiano prendendo rapidamente il sopravvento sulle tecnologie basate su VMs, anche grazie alla possibilità di eseguire VM in ambiente orchestrato, particolarmente in Kubernetes mediante un progetto quale KubeVirt. Pertanto, nel prosieguo si tenderà a focalizzarsi maggiormente su questa seconda classe di orchestratori.

Infatti, l'orchestrazione di containers si configura come una soluzione altamente flessibile. I containers offrono alle applicazioni basate su microservizi un'unità di deployment ideale delle applicazioni stesse e un ambiente di esecuzione autosufficiente: consentono di eseguire più parti di un'applicazione in modo indipendente nei microservizi, sullo stesso hardware, con un controllo superiore sui singoli componenti e cicli di vita. Pertanto, scegliere l'orchestrazione per gestire il lifecycle dei container aiuta inoltre i team DevOps a gestire più facilmente i flussi di integrazione e deployment continui (CI/CD). In concomitanza con l'utilizzo delle interfacce di programmazione delle applicazioni (API) e delle metodologie DevOps, i microservizi containerizzati rappresentano la base delle applicazioni cosiddette *cloud-native*, ossia specificamente progettate per essere eseguite in ambiente cloud. Infatti, le applicazioni della generazione precedente, lanciate entro VMs, erano semplicemente le stesse applicazioni che prima eseguivano sui server fisici, eseguite tuttavia in un ambiente virtuale. Di conseguenza non sfruttavano, se non minimamente, i vantaggi di scalabilità del cloud, se non a prezzo di notevoli artifizi (ad es., creazione di cluster di macchine per eseguire un'applicazione di tipo database).

Attraverso l'*automazione*, l'orchestrazione di container serve ad ottimizzare (e spesso a semplificare) la gestione di numerose attività, tra le quali citiamo le maggiormente importanti:

- Provisioning e deployment dei container, ciascuno logicamente incapsulato in un'unità elementare di schedulazione e orchestrazione detta *Pod*;

- Configurazione, pianificazione e assegnazione delle risorse di computing, storage, networking;
- Disponibilità (availability) dei container incapsulati in Pod;
- Scalabilità (*scale-out*) o rimozione (*scale-in*) dei container incapsulati nei Pod in funzione del bilanciamento dei workload dell'infrastruttura di computing, es. distribuzione e *scaling orizzontale* (scale-out/scale-in) dei carichi sul cluster di orchestrazione;
- Bilanciamento di carico (*load balancing* tra app/container in Pod che espletano pari servizio) e routing del traffico;
- Monitoraggio dell'integrità dei container incapsulati in Pod;
- Configurazione delle applicazioni in base al container/Pod in cui verranno eseguite;
- Protezione costante delle interazioni tra container/Pod.

Gli strumenti utilizzati per l'orchestrazione dei container offrono un framework per la gestione su larga scala delle architetture basate su container e microservizi. Tra i numerosi strumenti disponibili per la gestione del ciclo di vita dei container, Kubernetes, anche detto "K8s", ha attualmente raggiunto una maturità notevole, data la sua flessibilità ed estensibilità. Pensato per operare nello scenario datacenter (Cloud Computing tipico), Kubernetes è oggi considerato lo standard *de facto* per l'esecuzione di applicazioni containerizzate nel Cloud. K3s/Rancher e KubeEdge sono due architetture di ultima generazione che hanno lo scopo di adattare Kubernetes agli scenari del Fog/Edge Computing e, secondo le specifiche, supportano esattamente la stessa API di K8s.

3.2.1 Vantaggi e svantaggi delle soluzioni orchestrate

La tabella seguente illustra le caratteristiche degli orchestratori di containers, evidenziandone vantaggi e svantaggi.

Vantaggi	Svantaggi
Monitoring ✓ Data la loro flessibilità, è possibile automatizzare il deployment e la configurazione dei servizi virtuali. ✓ Inspection automatica sul ciclo di vita dei container (es., verifica dello stato di creazione, schedulazione, esecuzione, ecc., del servizio virtuale). ✓ Identificazione dei down-states (es., verifica dello stato di eventuale interruzione del servizio virtualizzato). ✓ Health-check sullo stato d'esecuzione (es., verifica che il servizio software risponda correttamente ad eventuali richieste).	**Complessità** × Infrastruttura molto più complessa, con ulteriori componenti software dedicate ai task di controllo, e agent di configurazione distribuiti tra le varie macchine, nonché potenziali nodi fisici aggiuntivi (es., per ospitare l'orchestratore). × Necessità di ridondare anche i nodi che ospitano l'orchestratore, nonché garantire la raggiungibilità degli stessi da parte dei nodi "orchestrati". × Necessità di monitoraggio continuo di raggiungibilità tra i nodi "orchestratori" e quelli "orchestrati", per consentire controllo e reazione

✓ Eventuali malfunzionamenti possono essere gestiti senza l'attesa del tempo di reazione umano grazie ad eventuali azioni intraprese automaticamente dall'orchestratore (es., rilanciare automaticamente un container andato in crash). **Auto-Scaling** ✓ Le repliche del servizio vengono gestite automaticamente in maniera ottimizzata, prendendo le decisioni di scale-out più opportune. ✓ Evita eventuali disservizi a seguito di aumenti di carico incrementando automaticamente le risorse allocate. ✓ Evita lo spreco di risorse riducendo quelle allocate quando il carico diminuisce. **Virtual Networking** ✓ Flessibilità di interconnessione tra i containers (microservizi). ✓ Semplicità nel definire e comporre nuove logiche di servizio. **Scheduling** ✓ Ottimizzazione dell'allocazione delle risorse sulle macchine fisiche disponibili. ✓ Possibilità di considerare, assumere e far rispettare numerosi vincoli, quali esigenze di risorse (CPU, memoria), disponibilità di storage, affinità tra task/microservizi e computing nodes, latenze e disponibilità di banda. ✓ Definizione dinamica dei parametri di ottimizzazione (objective functions) come minimizzazione dei costi, energia, load-balancing, ecc.	automatiche capillari sul respawn della funzione containerizzata (scheduling, provisioning ed esecuzione su nodo di computing più opportuno – "controllo adattivo"). × Manutenzione aggiuntiva sui componenti di orchestrazione. × Richiede personale altamente specializzato nella gestione dell'infrastruttura di orchestrazione. **Troubleshooting** × Potenzialmente più complicato, più strati software da controllare, particolarmente allorquando venga adottato un paradigma nel quale un servizio è implementato attraverso un numero elevato di micro-servizi. **Sicurezza** × Aumenta la superficie d'attacco, introducendo il problema della sicurezza infrastrutturale.

È importante notare che, nel caso di soluzioni orchestrate, il numero di guasti possibili è maggiore delle soluzioni presenti, a causa del maggior numero di componenti coinvolti (es., il nodo che ospita l'orchestratore); pertanto, se da un lato una soluzione orchestrata offre maggiori garanzie di mantenimento del servizio (es., spostando un servizio software su un altro nodo fisico), l'orchestratore stesso deve essere ridondato per prevenire guasti/attacchi miranti ad abbattere proprio questo componente.

3.2.2 Kubernetes

Kubernetes (abbreviato con *K8s*) è uno strumento open-source per l'orchestrazione dei *container* in origine ideato e sviluppato presso Google che, nel 2015, ne ha ceduto il controllo del progetto alla CNCF (Cloud Native Computing Foundation), allora di recente formazione. L'orchestrazione di Kubernetes consente di creare servizi applicativi che si estendono su più container Linux incapsulati nei Pod, programmare tali container/Pod in un cluster, gestirne la scalabilità e l'integrità nel tempo. Permette anche di eliminare molti dei processi manuali previsti dal deployment e dalla scalabilità di applicazioni containerizzate, nonché di gestire in modo efficiente e ottimizzato le risorse di cluster di host/computing node, sia fisici (server *bare-metal*) sia virtuali (*VM*), di supporto all'esecuzione dei container/Pod. Più in generale, Kubernetes garantisce l'affidabilità e la scalabilità necessarie ad adottare un'infrastruttura containerizzata negli ambienti di produzione: i cluster K8s possono espandersi su più host/computing node situati su cloud pubblici, privati o ibridi, ragion per cui Kubernetes è la piattaforma ideale per l'hosting di applicazioni *cloud-native* caratterizzate da elevata scalabilità. Infine, tale piattaforma semplifica il bilanciamento e la portabilità dei workload, in quanto permette di spostare le applicazioni tra diversi ambienti senza doverle riprogettare.

Kubernetes descrive e memorizza (attraverso metadati salvati in DB dedicato) lo *stato desiderato* per i container/Pod orchestrati, quindi si occupa di modificarne lo stato attuale per raggiungere quello desiderato a una velocità controllata secondo un workflow detto *rollout*. Per esempio, si può automatizzare Kubernetes per fargli creare ulteriori container/Pod per servizi specifici, rimuovere i container esistenti (mediante workflow di *rollback*) e/o adattare le loro risorse a quelle richieste dal nuovo container. Inoltre, possono essere realizzati componenti software personalizzati (detti *operatori*), integrabili in K8s, che osservano lo stato attuale del sistema e lo riconciliano con quello desiderato.

Il deployment di Kubernetes genera un cluster K8s. Esso è logicamente costituito da:

- *control-plane*, es. da uno o più nodi "orchestratore" detti *nodi master* (secondo configurazione HA), cui compete il controllo dell'infrastruttura di computing e del mantenimento dello stato desiderato;
- *data-plane*, es. da un insieme di nodi "orchestrati" detti *nodi worker*, cui compete l'esecuzione delle applicazioni containerizzate e incapsulate in Pod.

Ogni cluster K8s possiede almeno un master node e almeno un worker node. In pratica, come accennato, i nodi worker del data-plane ospitano i Pod che compongono i carichi di lavoro delle applicazioni containerizzate (workload applicativi). I nodi master componenti il control-plane del cluster, invece, gestiscono i nodi worker del data-plane dello stesso cluster e schedulano, allocano, fanno eseguire, scalano e ri-scalano i Pod che su essi eseguono. Negli ambienti di produzione, per ragioni di fault-tolerance e high-availability (alta disponibilità di servizio ottenuta attraverso ridondanza), di solito il control-plane è costituito da più nodi master ridondati, ciascuno eseguito su

un host (server fisico o server virtuale/VM che sia) ad esso dedicato; di norma, poi, sul cluster insistono molteplici e numerosi nodi worker.

Kubernetes implementa un ambiente di esecuzione molto modulare, infatti sia control-plane sia data-plane sono suddivisi in componenti funzionali, concettualmente rappresentabile come nella Figura 1 seguente.

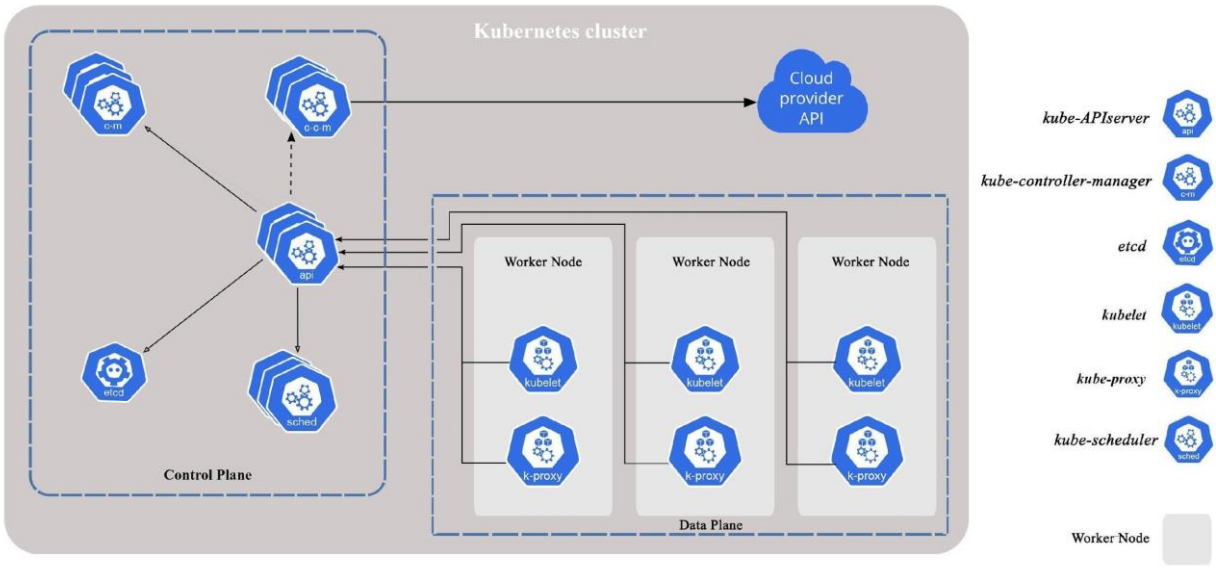

FIGURA 1 – COMPOSIZIONE MODULARE DI *CONTROL-PLANE* E *DATA-PLANE* IN KUBERNETES.

I componenti del control-plane assumono decisioni globali sul cluster (es., scheduling e spawn dei Pod sui worker node), oltre a rilevare e rispondere agli eventi di esso (es., avvio del numero necessario di repliche di un Pod quando il numero di esse stabilito per un dato deployment non sia ancora soddisfatto). I componenti del control-plane possono essere eseguiti separatamente l'uno dall'altro su qualsiasi host (master node) del cluster; tuttavia, per semplicità, la configurazione default generalmente li avvia tutti sullo stesso host (nodo master) e prevede di non eseguire i container/Pod delle applicazioni utente su tale nodo (bensì su altri nodi che siano worker node, cioè quelli che costituiscono il data-plane). Essi sono:

- *kube-apiserver*: è l'implementazione del server API, componente che espone l'API K8s e costituisce il front-end del suo control-plane, progettato per scalare orizzontalmente (scale-out), cioè esso scala deployando ed eseguendo più istanze diverse di sé, per le quali è automaticamente attivato il load-balancing del traffico ad esse destinato.
- *etcd*: DBMS che memorizza e mantiene lo stato del cluster Kubernetes, in generale attraverso un numero di nodi distribuiti, grazie ad un meccanismo basato su un DB schema key/value che garantisce coerenza/consistenza e alta disponibilità di tutti i metadati relativi allo stato del cluster. I dati vengono inoltre salvati in caso di DR (disaster recovery).

- *kube-scheduler*: controlla l'allocazione dei Pod appena creati e non ancora assegnati ad alcun worker node, selezionando perciò un nodo worker sul quale eseguirli; considera i seguenti fattori per assumere le decisioni di scheduling: requisiti di necessità risorse individuali e collettive, vincoli hardware / software / policy, specifiche di affinità e anti-affinità, località dei dati, interferenza tra workload diversi e deadline di essi.
- *kube-controller-manager*: esegue i processi controller quali *Node Controller* (responsabile di rilevare e reagire quando i nodi worker vanno in stato down), *Replication Controller* (responsabile del mantenimento del numero corretto/necessario di Pod nel sistema), *Endpoint Controller* (fa in modo che determinati Pod diventino endpoint per servizi esposti), *Service Account Controller* & *Token Controller* (creano, rispettivamente, account predefiniti per la gestione del sistema e token di accesso alle API K8s).

I componenti del data-plane, invece, forniscono a Kubernetes l'ambiente di runtime, mantengono i Pod in esecuzione e vengono eseguiti su ogni nodo worker. Essi sono:

- *kubelet*: agent eseguito su ogni nodo worker del cluster, si assicura che i container siano incapsulati e in esecuzione nei Pod; esso accetta un set di PodSpec (insieme di specifiche funzionali, fornito tramite vari meccanismi) e garantisce che i container descritti in tali PodSpec siano integri e in esecuzione; non gestisce comunque i container, eventualmente presenti sul nodo worker, che non siano stati creati da Kubernetes.
- *kube-proxy*: network proxy eseguito su ogni nodo worker del cluster, mantiene le network rules su tutti tali nodi, che consentono l'instaurazione di sessioni interne e/o esterne al cluster verso i Pod/endpoint (es., il networking Pod/Pod interni e Pod/end-system esterni al cluster K8s). Utilizza il packet filtering a livello kernel di SO (es. *netfilter*) se disponibile, altrimenti provvede esso stesso all'inoltro del traffico.
- *Container Runtime Engine*: come già visto, è il software responsabile dell'esecuzione dei container; Kubernetes supporta diversi CRE, quali *Docker*, *containerd*, *CRI-O* e qualsiasi implementazione di *Kubernetes CRI* (Container Runtime Interface).

Inoltre, il Cluster Networking (comunicazioni intra-cluster e inter-cluster) è una parte essenziale di Kubernetes, comunque di non facile comprensione e implementazione, anche nel comportamento atteso. Kubernetes è basato sulla condivisione di host (fisici e/o virtuali) tra applicazioni containerizzate: ciò richiede la garanzia che due Pod/container/applicazioni non provino a utilizzare le stesse porte. In tale scenario, il coordinamento dell'assegnazione delle porte tra più applicazioni può essere molto difficile da compiere su larga scala, potenzialmente esponendo gli utenti a issue riscontrabili a livello di cluster, che siano quindi al di fuori del loro controllo. L'allocazione dinamica delle porte determina molte complicazioni al sistema complessivo: ad esempio, ogni container/applicazione dovrebbe considerare la porta assegnatagli come un flag distintivo, i server API dovrebbero sapere come inserire numeri di porta dinamici nelle configurazioni, i servizi dovrebbero sapere come scoprirsi (discovery) a vicenda, ecc. Piuttosto che affrontare questo problema, Kubernetes adotta un approccio diverso che consiste nell'assegnare a ciascun Pod un

proprio indirizzo IP, per cui non è necessario creare collegamenti espliciti tra Pod e non è quasi mai necessario occuparsi del mapping delle porte dei container/applicazioni alle porte dell'host. Questo approccio crea un modello pulito e retrocompatibile, in cui i Pod possono essere trattati in modo molto simile a VM/host fisici dal punto di vista dell'allocazione delle porte, del naming, della discovery dei servizi, del load-balancing, della configurazione e migrazione/riallocazione dei container/applicazioni tra host/nodi del cluster.

Tale modello è non solo complessivamente meno complicato ma è principalmente compatibile con l'obiettivo di Kubernetes di consentire il porting a basso impatto delle applicazioni dalle VM ai container. Se il workload veniva precedentemente eseguito in una VM, la VM possedeva un indirizzo IP e poteva comunicare (secondo policy) con altre VM dell'infrastruttura virtualizzata. Ciò rappresenta lo stesso modello di base implementato in Kubernetes: gli indirizzi IP esistono nell'ambito del Pod, cioè eventuali container multipli (detti *sidecar*) incapsulati all'interno di un Pod condividono lo stack TCP/IP, naturalmente inclusi i loro indirizzi IP e MAC. Questo concetto implica che i sidecar container incapsulati all'interno di uno stesso Pod possano agilmente raggiungere le porte l'uno dell'altro sul "localhost" e che debbano sì coordinare l'utilizzo delle porte ma ciò, concettualmente, non differisce dall'allocazione delle porte ai processi nel SO guest di una VM. Questo modello è detto "IP-per-Pod": il modo in cui sia esso sia il *port-forwarding* vengano implementati sono dettagli peculiari del particolare CRE (es. Docker, containerd, CRI-O, ecc.) in uso. È possibile richiedere/allocare porte sull'host del nodo worker stesso dette *porte host* (HostPort o NodePort) che facciano forwarding alle porte del Pod dette *porte target* (TargetPort) ma questo è uno scenario di nicchia, infatti il Pod stesso è intrinsecamente cieco all'esistenza (o meno) delle porte host.

Nel corso del tempo sono state rilasciate varie soluzioni implementative delle specifiche richieste per la comunicazione dei container intra-cluster e inter-cluster. L'acronimo *CNI* (*Container Network Interface*) indica uno standard atto a facilitare/normalizzare la configurazione dello strato di networking per i container eseguiti mediante un CRE. Nell'ecosistema Kubernetes sono quindi disponibili molteplici plug-in CNI ciascuno dei quali, nonostante le funzionalità di base comuni a tutti, integra feature peculiari (es. security, agilità, essenzialità): è necessario (non opzionale) sceglierne uno secondo tali funzionalità e secondo le esigenze dello scenario di deployment del cluster K8s. Il plug-in CNI è quindi prodromico al deployment del cluster, fa sì che siano soddisfatti tutti i requisiti di rete e siano implementate tutte le feature di internetworking che il cluster K8s necessita. I CRE offrono diverse modalità di networking, ognuna per scopi differenti, come ad esempio "None" (nessuna connettività di rete), "Host" (connessione diretta allo stack di rete TCP/IP dell'host), "Default Bridge" (creazione di un network bridge e interconnessione del container tramite un indirizzo IP dedicato - modalità default in Docker), "Custom Bridge" (creazione di un bridge user-defined con funzionalità aggiuntive, quali VLAN, isolamento, ecc.). Inoltre, al di là delle modalità di networking, è possibile implementare soluzioni multi-host che sfruttino il concetto di *overlay network*. L'idea alla base dell'iniziativa standard CNI (specifica *appc/CNI*), quindi, è proprio

quella di fornire un framework per il provisioning e l'allocazione dinamica delle configurazioni di rete dei container, nell'ottica della loro orchestrazione all'interno di un cluster. Il plug-in CNI è quindi responsabile della creazione dell'interfaccia di rete virtuale (Virtual NIC *vethX*), dell'assegnazione dell'indirizzo IP, delle funzionalità di connettività, ecc. durante l'intero ciclo di vita del container. In questo contesto, all'interno del cluster Kubernetes, questa metodologia consente al componente *kubelet* prima visto di configurare automaticamente lo strato di networking per i Pod (*Pod network*) semplicemente invocando il plug-in CNI installato nel cluster.

In definitiva, in un cluster K8s è obbligatorio il deployment di un plug-in CNI che "costruisca" le Pod network in modo che i container/Pod del cluster siano abilitati a scambiare traffico di rete tra loro e con l'esterno.

3.2.3 K3s/Rancher

Rancher è un progetto open-source recente che consente la gestione di più cluster Kubernetes separati [Rancher], contraddistinto da limitati requisiti computazionali. Nello specifico, Rancher fornisce un control-plane centralizzato nonostante la presenza sottostante di più cluster potenzialmente indipendenti, ciascuno dei quali controllato localmente da una istanza separata di Kubernetes.

Rancher è una soluzione fin da subito orientata all'eterogeneità che contraddistingue l'Edge layer della rete, per cui si propone di consentire e supportare l'installazione di un'istanza di Kubernetes su qualunque "cluster" di risorse, anche non convenzionali (ad esempio i RAN server sulle reti d'accesso dei providers, dispositivi con capacità di computing ridotte, ecc.). Il fatto che Kubernetes sia stato pensato per operare su datacenters pone numerose sfide al suo deployment su piccoli clusters tipici dell'Edge computing: ad esempio, il solo control-plane di Kubernetes "vanilla" potrebbe richiedere una quantità di risorse paragonabile a quelle totali disponibili sul cluster. Per questa ragione, congiuntamente alla soluzione architetturale multi-cluster, Rancher propone K3s [RancherK3s], anche noto come *"The Lightweight Kubernetes Distribution Built for the Edge"*: una distribuzione "alleggerita" di Kubernetes, facile da installare e pensata per scenari "resource constrained" (nodi di computing con risorse limitate) e "low touch operations" (operations non invasive o dove non agevoli sul campo). Alcuni scenari e ambienti d'impiego particolarmente adatti a K3s sono l'Edge computing, le architetture ARM, i device/appliance IoT e la CI (Continuous Integration, nell'ambito dello sviluppo agile del software).

K3s/Rancher costituisce una possibile soluzione al problema dell'interoperabilità tra cluster diversi che, potenzialmente, possono operare anche in maniera autonoma. La presenza di una istanza completa di Kubernetes (o K3s) su ciascun cluster, infatti, garantisce la possibilità che questo possa "auto-orchestrarsi" nell'eventualità in cui il cluster risulti impossibilitato a comunicare con i suoi peer. Di contro, il carico computazionale richiesto su ogni "mini-cluster" potrebbe risultare comunque eccessivo.

Si introduce dunque una rappresentazione grafica schematica in Figura 2, che condensa i punti di vista architetturale e funzionale dei componenti la soluzione di orchestrazione K3s/Rancher, seguita dalla descrizione di essi.

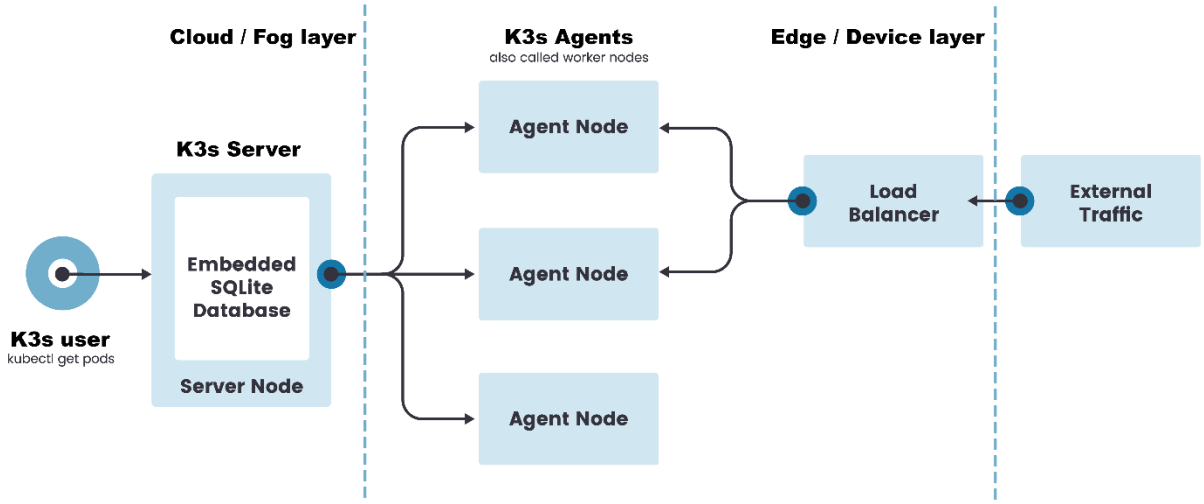

FIGURA 2 – COMPOSIZIONE ARCHITETTURALE E FUNZIONALE MODULARE DI K3S/RANCHER.

Cloud / Fog layer:

- K3s *Server* Node, c.d. K3s *Master* Node (control-plane):
  - K3s *API Server*: convalida e configura i dati per gli oggetti Pod, servizi, replication controller, ecc. Offre operazioni REST e il front-end dello stato condiviso del cluster;
  - *SQLite3* DBMS: default datastore dello stato del cluster K8s usato per disaster recovery, costituito da un DB key/value che garantisce coerenza/consistenza (ed eventualmente alta disponibilità) di tutti i metadati relativi allo stato del cluster;
  - *K3s agent* process: processo di servizio speciale utile alle operazioni di join/configurazione iniziali, in esecuzione su entrambe le tipologie di Node, *Server* e *Agent*. Nel caso del K3s *Server* Node, agisce come *server process* in attesa delle connessioni WebSocket aperte dalle controparti (K3s agent process client-side) in esecuzione sui K3s *Agent* Node.

In questa configurazione base, ogni K3s *Agent* Node è registrato/autenticato presso lo stesso K3s *Server* Node. L'amministratore di sistema (K3s *user*) può manipolare le risorse del cluster K3s invocando il K3s *API server* (in esecuzione sul K3s *Server* Node) mediante i comandi usuali necessari a configurare, monitorare e controllare un tipico cluster Kubernetes di risorse orchestrate.

Edge / Device layer:

- K3s *Agent* Node, c.d. K3s *Worker* Node (data-plane):

- *K3s agent* process: processo di servizio speciale utile alle operazioni di join/configurazione iniziali, in esecuzione su entrambe le tipologie di Node, *Server* e *Agent*. Nel caso dei K3s *Agent* Node, agisce come *client process* che apre una connessione WebSocket verso la controparte (K3s agent process server-side) mediante la quale i K3s *Agent* Node si registrano al K3s *Server* Node. Tale connessione (client-side) è mantenuta da un load-balancer client-side integrato nel processo K3s agent.

La soluzione K3s/Rancher consente dunque di gestire cluster medio-grandi all'Edge layer al prezzo di maggior overhead computazionale, richiedendo la presenza di una istanza "master" di Kubernetes in ciascuno di tali cluster. Per cui, può risultare idoneo a scenari in cui è necessario che i singoli cluster siano in grado di operare indipendentemente dalla loro capacità di comunicare con un'istanza "master" remota (ad es. in Cloud layer). Infatti, in assenza di connettività o in condizioni non ottimali di comunicazione, il cluster K3s all'Edge layer risulterebbe comunque dotato di un control-plane che possa adempiere i task di orchestrazione.

3.2.4 KubeEdge

KubeEdge è un sistema open-source piuttosto "giovane" - e ancora poco maturo - utile a estendere le capacità Kubernetes di orchestrazione delle applicazioni nativamente containerizzate agli host device IoT che si trovano all'Edge layer [KubeEdge]. È infatti ideato, basato e costruito su Kubernetes e fornisce supporto infrastrutturale fondamentale per networking, app, distribuzione e sincronizzazione dei metadati (relativi allo *stato desiderato* del cluster) tra Cloud layer ed Edge layer. L'obiettivo dell'impiego di KubeEdge è quello di sfruttare una piattaforma aperta per abilitare l'Edge computing. Il progetto è stato recentemente incubato presso la CNCF (Cloud Native Computing Foundation), per cui è abbastanza ragionevole attenderne rapide evoluzioni, con frequenti rilasci che maturino la tecnologia e integrino eventuali nuove estensioni.

KubeEdge supporta e sfrutta anche il protocollo di comunicazione MQTT (Message Queue Telemetry Transport), standard ISO che poggia sullo stack TCP/IP e si basa sul pattern *publish-subscribe*, progettato per gli scenari d'impiego in cui è richiesto un basso overhead protocollare (es., basso impatto computazionale dovuto al protocollo) e dove il throughput (larghezza di banda) utile alle comunicazioni può essere limitato, esattamente come può avvenire nel caso di device IoT all'Edge layer. Dato il pattern publish-subscribe implementato da MQTT, KubeEdge richiede la presenza di un *message broker* nei propri nodi di computing posti all'Edge, un servizio software aggiuntivo responsabile della ricezione/distribuzione dei messaggi applicativi (ad es. sincrofasori provenienti da PMU/PDC) provenienti dai device IoT che rilevano le metriche di interesse.

KubeEdge consente anche agli sviluppatori di creare logica applicativa personalizzata secondo il dominio di interesse, e abilita la comunicazione con *resource constrained IoT device* (device hardware general-purpose a basso costo piuttosto limitati in termini di risorse, quali Raspberry-Pi et similia) all'Edge layer.

È dunque utile introdurre una rappresentazione grafica schematica in Figura 3, che condensa i punti di vista architetturale e funzionale dei componenti la soluzione di orchestrazione KubeEdge, seguita dalla descrizione di essi.

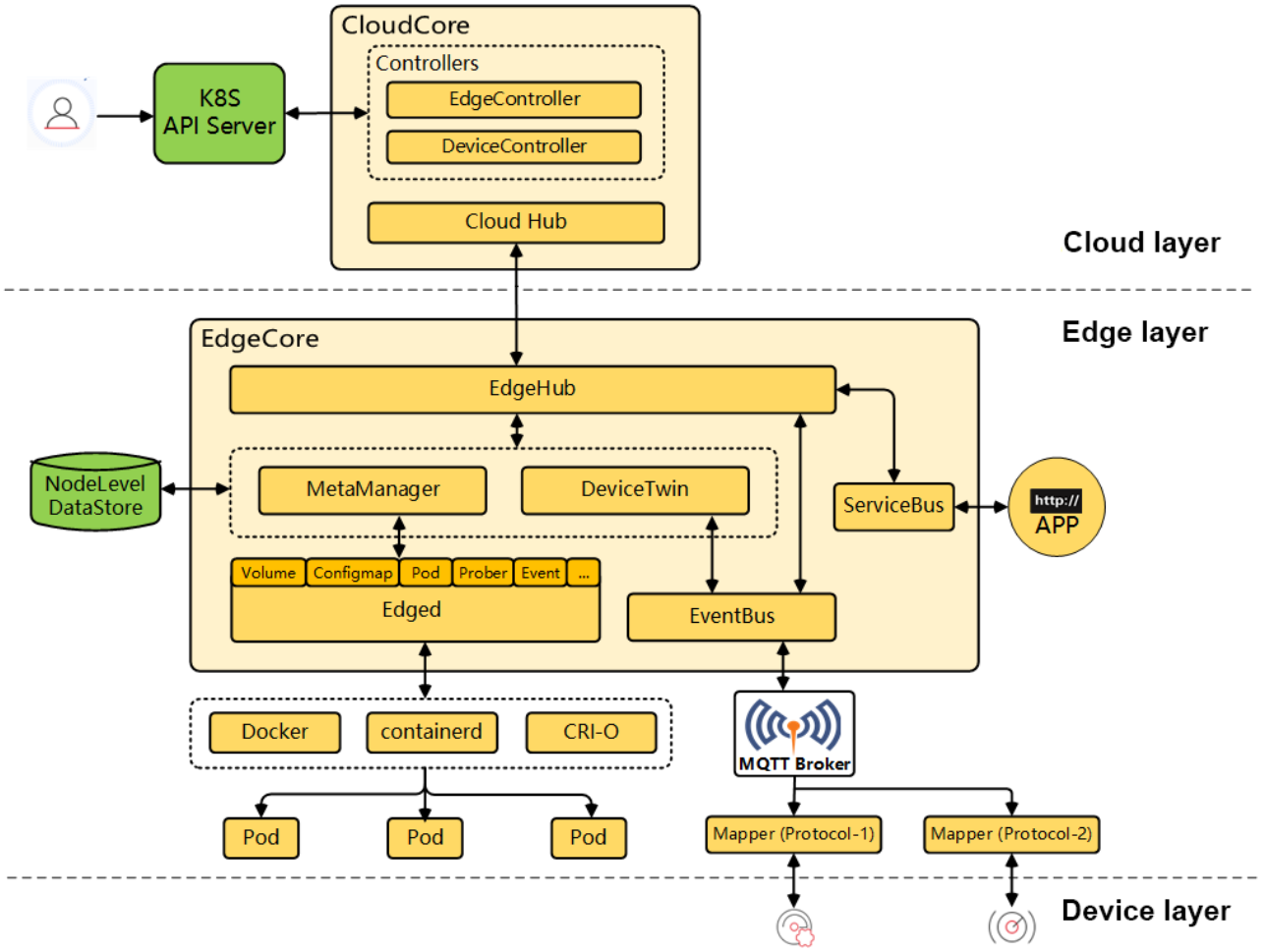

FIGURA 3 – COMPOSIZIONE ARCHITETTURALE E FUNZIONALE MODULARE DI KUBEEDGE.

Cloud layer:

- K8s *API Server*: è esattamente il modulo *kube-apiserver* già visto nella trattazione di Kubernetes vanilla; esso convalida e configura i dati per gli oggetti Pod, servizi, replication controller, ecc. Offre operazioni REST e il front-end dello stato condiviso del cluster;
- KubeEdge *CloudCore*:
  - *EdgeController*: K8s controller esteso che gestisce i metadati dei nodi di computing all'Edge (Worker Node) e dei Pod su essi eseguiti, con l'obiettivo di selezionare lo specifico nodo di computing all'Edge (Worker Node) cui indirizzare dati. Esso assolve la funzione di bridging tra il K8s *API Server* al Cloud layer e l'*EdgeCore* in esecuzione su ciascun Edge Worker Node all'Edge layer, agendo nei confronti dell'*EdgeCore* sia come "Downstream Controller" (nel

monitorare le azioni/gli eventi di configurazione - aggiunta/aggiornamento/eliminazione - delle risorse presenti all'Edge layer compiute dall'amministratore del sistema sull'*EdgeCore* tramite il K8s *API Server* e nell'inviare all'*EdgeCore* - tramite *CloudHub* - gli opportuni messaggi di aggiornamento di tali risorse per sincronizzarne lo stato tra Cloud layer ed Edge layer a seguito degli eventi/delle azioni di configurazione, cioè effettuare il *publish* degli eventi di aggiornamento delle risorse verso l'*EdgeCore*) sia come "Upstream Controller" (nel ricevere dall'*EdgeCore* i messaggi di aggiornamento dello stato delle risorse presenti all'Edge layer e sincronizzarli con il K8s *API Server* al Cloud layer, cioè nel sincronizzare lo stato osservato e aggiornato delle risorse e degli eventi all'Edge layer con il K8s *API Server*, avendo previamente effettuato il *subscribe* agli eventi di aggiornamento dall'EdgeCore);
  - *DeviceController*: modulo responsabile della gestione degli IED/end-device IoT, implementata utilizzando le K8s *Custom Resource Definitions* (CRD) per descrivere sia metadati/stato degli end-device IoT, sia il controller che ne sincronizza gli update tra Edge layer e Cloud layer. Per implementare la gestione dei dispositivi, fa uso delle risorse *DeviceModel* (descrizione/specifica statica delle proprietà del dispositivo da questo esposte ai lettori di esse per accedervi, rappresenta un template riutilizzabile che consente di creare e gestire molti dispositivi) e *DeviceInstance* (oggetto dispositivo effettivo, rappresenta un'istanza del DeviceModel che fa riferimento alle proprietà definite in quest'ultimo; contiene dati che cambiano dinamicamente, quali lo stato attuale riportato dal dispositivo e lo stato desiderato di una proprietà di esso);
  - *CloudHub*: server WebSocket responsabile della comunicazione, presentazione e visualizzazione delle variazioni di stato sul Cloud layer, del caching e dell'invio/ricezione dei messaggi di sincronizzazione alla/dalla controparte client *EdgeHub* (appartenente all'*EdgeCore* in esecuzione su ciascun Edge Worker Node all'Edge layer). Più in generale, mediante la connessione con *EdgeHub*, esso agisce da modulo intermediario tra i *Controller* suddetti (con i quali comunica direttamente internamente al *CloudCore*) e l'Edge layer. La connessione del *CloudCore* all'*EdgeCore* (per tramite di *CloudHub* con la controparte client *EdgeHub*) avviene attraverso il protocollo WebSocket che incapsula i messaggi HTTP contenenti le informazioni di syncing (sincronizzazione). La funzione fondamentale di *CloudHub* è dunque quella di abilitare e mantenere un canale di comunicazione stabile tra l'*EdgeCore* posto all'Edge layer e i *Controller* del *CloudCore* posti al Cloud layer.

Edge layer:

- DataStore: DBMS di back-end che offra la persistenza eventualmente richiesta ai dati applicativi memorizzati localmente al nodo Edge (3rd-party, non fornito da KubeEdge);
- KubeEdge *EdgeCore*:
  - *EdgeD*: agent in esecuzione sui nodi di computing all'Edge layer (Edge Worker Node), gestisce il ciclo di vita dei Pod che incapsulano le relative applicazioni containerizzate

(agendo sul CRE) e consente agli amministratori di sistema di distribuirne i workload nei nodi stessi;

- *EdgeHub*: client WebSocket responsabile, per il computing dei Worker Node all'Edge layer, dell'interazione di *EdgeCore* con *CloudCore*, cioè della sincronizzazione degli aggiornamenti delle risorse tra Edge layer e Cloud layer, della segnalazione al Cloud layer di modifiche di stato degli IED/end-device IoT (Device layer update) e dei Worker Node/nodi di computing Edge (Edge layer update). Più in dettaglio, rappresenta il componente con cui l'*EdgeCore* dell'Edge Worker node si interfaccia con il componente controparte *CloudHub* del *CloudCore* che esegue sul Cloud Master Node presente nel Cloud layer. Funge cioè da link di comunicazione tra Edge layer e Cloud layer, inoltrando i messaggi ricevuti dal Cloud layer al modulo corrispondente all'Edge layer e viceversa;
- *MetaManager*: message processor tra *EdgeD* ed *EdgeHub*, effettua store/retrieve dei metadati in/da database mediante un DBMS locale al nodo Edge (es., lightweight SQLite);
- *DeviceTwin*: memorizza lo stato degli end-device IoT e lo sincronizza con il Cloud layer, gestisce gli attributi degli end-device IoT, crea la membership tra l'end-device e il nodo Edge, fornisce la query interface per le applicazioni;
- *EventBus*: client MQTT che interagisce con i server MQTT (Mosquitto, che possono insistere sugli end-device o sui nodi Edge) che offrono funzionalità di publish/subscribe per la comunicazione asincrona tra edge-side ed end-device.

- *MQTT Broker*: *publish-subscribe* message broker responsabile della distribuzione dei messaggi (es., synchrophasor data-stream) ai dispositivi IoT/nodi Edge (Worker Node).

In questa configurazione generale, valevole per qualsiasi deployment della soluzione, ogni KubeEdge *Worker* Node (nodo di computing all'Edge layer) è registrato/autenticato presso uno stesso KubeEdge *Master* Node che agisce da control-plane centralizzato. Attraverso l'interazione con il front-end di quest'ultimo (cioè invocando il K8s *API Server* in esecuzione sul KubeEdge *Master* Node) posto nel Cloud layer, l'amministratore di sistema può manipolare le risorse del cluster mediante i comandi usuali necessari a configurare, monitorare e controllare un tipico cluster Kubernetes di risorse orchestrate (Pod e relative applicazioni containerizzate / workload).

La soluzione KubeEdge, rispetto a K3s/Rancher, permette di gestire piccoli cluster all'Edge layer con minor overhead computazionale, dato che non richiede la presenza di una istanza "master" di Kubernetes in ciascuno di essi. Di contro, può non risultare idoneo a scenari in cui è necessario che i singoli cluster siano in grado di operare indipendentemente dalla loro capacità di comunicare con un cluster "master" ubicato in un sito remoto (ad es. in Cloud layer). Infatti, in assenza di connettività o in condizioni non ottimali di comunicazione, i cluster all'Edge layer risulterebbero privi di un control-plane che possa espletare i task di orchestrazione.

Segue una tabella di confronto tra le due principali soluzioni di orchestrazione orientate a scenari Edge Computing, dove vengono identificati e schematizzati vantaggi e svantaggi di ciascuna.

Rancher K3s	KubeEdge
Vantaggi ✓ Distribuzione Kubernetes certificata, sviluppata eliminando da K8s componenti poco utilizzati e sacrificando le caratteristiche di scalabilità (es. database SQLite anziché etcd). ✓ Al presente, tecnologia abbastanza matura in fase stabile di sviluppo, derivando da K8s vanilla, da poco supporta anche la configurazione multi-master (ridondanza dei nodi che ospitano le entità control-plane). ✓ Leggera e semplificata (stripped-down), creata per IoT ed Edge Computing: in virtù del software minimale espone ridottissima superficie d'attacco. ✓ Progetto completamente open-source, per il quale è disponibile anche il supporto commerciale / enterprise di Rancher Labs. ✓ Architettura concettuale, core feature e strumenti admin fully-compliant con K8s, mutuati interamente da K8s nell'organizzazione e gestione del cluster di nodi master e nodi worker, per cui di semplice utilizzo per tecnici ICT già esperti di K8s (non richiede competenze aggiuntive). ✓ Orientata alla High Availability (HA) e progettata per workload di produzione in luoghi remoti non presidiati, con risorse limitate o all'interno di appliance IoT. ✓ Il package consiste di un singolo file binario di dimensioni molto ridotte (<40MB), che semplifica e riduce le dipendenze e i passaggi necessari per installare, eseguire e aggiornare automaticamente un cluster K3s di produzione. ✓ Memory footprint contenuta (<1GB), sia per nodi master sia per nodi worker. ✓ Il funzionamento di tutti i componenti del control-plane derivati da Kubernetes è incapsulato in un unico file binario che esegue come unico processo: ciò consente a K3s di automatizzare e gestire complesse operations sul cluster (es. distribuzione di certificati). ✓ Supporta architetture di processing eterogenee: x86-32bit/64bit, ARM 32bit/64bit.	**Vantaggi** ✓ Progetto completamente open-source: rilasciato e disponibile il codice sorgente di entrambi i moduli core (EdgeCore e CloudCore), di recente incubazione presso la CNCF. ✓ Basato sull'infrastruttura Kubernetes per gestire l'intero cluster: nodi di computing, applicazioni containerizzate e dispositivi. ✓ Scalabilità tipica di Kubernetes, in virtù dell'orchestrazione di applicazioni containerizzate orientate ai microservizi. ✓ Ottimizzato per scenari resource constrained all'Edge layer: il modulo EdgeCore può eseguire su nodi di computing con esigue risorse hardware (es. a bordo di device quali Raspberry-Pi). ✓ Supporta architetture di processing eterogenee: x86-32bit/64bit, ARM 32bit/64bit. ✓ Il control-plane (K8s master puro + modulo CloudCore in esecuzione sul nodo master del cluster) è sia cross-platform, sia cloud-agnostic: può eseguire sui cloud pubblici di tutti i cloud provider, sui cloud privati e sui cloud ibridi. ✓ Il data-plane (modulo EdgeCore agente in esecuzione sui nodi worker del cluster) implementa la modalità off-line: continua a eseguire anche se disconnesso dal control-plane (modulo CloudCore). ✓ Offre la possibilità di controllare dal Cloud layer (in remoto) il software in esecuzione sul data-plane (es., sui nodi worker in esecuzione sugli Edge device). ✓ Supporta il data management (gestione dei dati) e un motore di pipeline per data analytics. **Svantaggi** × Tecnologia ancora relativamente poco matura, in piena fase di sviluppo. × Attualmente, documentazione ufficiale confusa, troppo essenziale, scarsa e non sempre attendibile.

- ✓ Ottimizzato per architetture di processing ARM: supportate sia ARMv7, sia la più recente ARM64 con binari e immagini multi-arch disponibili per entrambe, quindi particolarmente adatto al deployment su device quali Raspberry-Pi.
- ✓ Backend di archiviazione dello stato (metadati) del cluster leggero basato su SQLite3 quale DBMS di storage predefinito. Offre ancora compatibilità con i noti etcd3 (default per K8s), MySQL, PostgreSQL.
- ✓ Supporta Edge device autonomi: il data-plane può essere autonomo, anche utilizzabile senza connessione esterna verso il control-plane.

Svantaggi

- × Dal punto di vista del processing, implementa un data-plane più massiccio rispetto a KubeEdge: in quest'ultimo, infatti, si deploya solo un agent e non un worker node "completo" in stile K8s.
- × Non offre la possibilità di controllare dal cloud (in remoto) il software in esecuzione sul data-plane (es. sui nodi worker in esecuzione sugli Edge device).
- × Docker, se scelto quale Container Runtime Engine in luogo del default *containerd*, è noto causare problemi durante l'esecuzione di K3s, soprattutto se installato tramite package snap.
- × Eseguire *iptables* in modalità nftables invece che legacy genera instabilità all'esecuzione di K3s: necessario assicurarsi sempre di eseguire versioni iptables recenti per evitare incompatibilità.
- × Installazione e configurazione non semplici e immediate: i moduli CloudCore ed EdgeCore già compilati in eseguibili possono essere installati solamente su distribuzioni Linux Ubuntu e CentOS per architetture x86, amd64; per altre distro/arch. devono essere compilati dai sorgenti.
- × Sviluppo basato su SDK per l'aggiunta di dispositivi, distribuzioni di applicazioni, ecc., apparentemente semplificato ma non agevole.
- × Manutenzione: rollout (aggiornamento) e/o rollback (ripristino di una versione precedente) dei componenti software della soluzione di orchestrazione (infrastruttura di orchestrazione) e delle applicazioni containerizzate non sono semplici/agevoli né immediati in quanto tale manutenzione richiede l'aggiornamento sia dei componenti KubeEdge, sia del cluster K8s.
- × Per rendere efficace l'orchestrazione, il data-plane (es., i nodi worker in esecuzione sugli Edge device) può tollerare solo temporaneamente la mancanza di connettività alla rete del control-plane (es., al nodo master K8s controllante).
- × Il data-plane (pool di nodi worker del cluster all'Edge layer) deve necessariamente prevedere la presenza di un control-plane (o intero cluster) K8s già esistente o da installare ad hoc (in tale ultimo caso, procedura non banale).
- × Il tool di installazione (quando non condotta tramite compilazione di source code) e amministrazione *keadm* è ufficialmente rilasciato solo per le distro Ubuntu e CentOS.
- × Non supporta Edge device autonomi: data-plane non autonomo, non utilizzabile senza connessione esterna verso il contro-plane benché ne tolleri temporanea disconnessione.
- × Non supporta l'installazione di network plugin arbitrari (es., *Cilium*, *Calico*, *Flannel*); non è pertanto possibile creare dei servizi Kubernetes (es., "ClusterIP", "NodePort") che siano validi per tutto il cluster.

4 Casi d'uso

Questa sezione analizza i casi d'uso che sono stati presi a riferimento per valutare le possibili differenze di implementazione del servizio attraverso le varie tecnologie menzionate nel capitolo precedente, ossia mediante l'utilizzo di dispositivi fisici dedicati (scenario "classico"), oppure attraverso la virtualizzazione di essi e la loro esecuzione su piattaforme dotate di supporto alla virtualizzazione. In quest'ultimo scenario, verrà distinto il caso in cui le piattaforme virtualizzate siano "stand-alone" dal caso in cui esse siano "orchestrate" da un sistema esterno.

Dopo la descrizione dell'architettura e dei componenti funzionali di processo elettro-energetico coinvolti in ciascun caso d'uso, vengono identificati i Key Performance Indicators (KPIs) dei relativi servizi che porteranno a individuare e caratterizzare, negli sviluppi futuri dell'attività di ricerca, le potenziali criticità di questi componenti dal punto di vista ICT.

4.1 Principali dispositivi coinvolti

In questa sotto-sezione si descrivono perciò i principali dispositivi fisici del mondo elettro-energetico coinvolti nella generazione / elaborazione dei dati critici per i casi d'uso considerati nella presente attività di ricerca.

4.1.1 Phasor Measurement Unit (PMU)

Nello scenario della power grid attuale, la *Phasor Measurement Unit* (PMU, IEEE-C37.118.1-2011 [PES2011, PES2018]) è un IED (*Intelligent Electronic Device*), cioè un device fisico special-purpose tipicamente equipaggiato con software minimale, specifico dei dispositivi embedded che offre supporto all'esecuzione del processo di acquisizione dati di misura (funzione software potenzialmente mission-critical, secondo il particolare scenario di deployment delle PMU stesse: distribution grid oppure transmission grid).

Una PMU integra sensoristica on-board collegata a nodi della power grid attuale (quali battery lead/lithium storage, Direct Current supply, R-L-C load, H&H load, MCI, Smart Home TWM, ecc.); le misure da essa prodotte sono dette *sincrofasori* poiché sincronizzate nel tempo grazie ad un sistema di riferimento temporale, solitamente GPS. Tra le caratteristiche principali delle PMU si hanno: (1) elevata frequenza di misura, (2) precisione, (3) bassa latenza, (4) resilienza ad interferenze esterne (ad esempio le armoniche).

Essi vengono installati in maniera distribuita sulle reti elettriche MT e AT per misurare le tensioni nodali e le correnti di linea, ma anche la frequenza e il Rate of Change of Frequency (ROCOF), ossia

il tasso di variazione di frequenza nel tempo. Le misure rilevate entrano nel Wide Area Monitoring Systems (WAMS), consentendo un monitoraggio real-time delle condizioni della rete[2].

4.1.2 Phasor Data Concentrator (PDC)

A un livello superiore rispetto ai dispositivi PMU attualmente dispiegati nella power grid si trovano i componenti funzionali Phasor Data Concentrator (PDC, IEEE C37.244-2013 [PES2013]) che, differentemente dal caso del device PMU, vengono eseguiti su dispositivi hardware general-purpose (server tradizionali) come normali processi, spesso su un sistema operativo Windows. Questi raccolgono i dati provenienti da molteplici PMU, aggregandoli e rendendoli disponibili in maniera più astratta e funzionale a tutte le applicazioni di monitoraggio e di controllo di più alto livello che necessitano di tali dati, quali ad esempio visualizzazione delle informazioni, allarmi, analisi sofisticate, funzioni di controllo o di automazione (Figura 4).

Le serie temporali dei sincrofasori (aggregati) raccolti da un PDC possono essere processate e inviate ad altri PDC come unico data-stream, quindi utilizzate per varie applicazioni quali dashboard (UI visualizzazione di info/allarmi), storage in time-series DB, on-line/off-line analytics, secondo la modalità operativa del PDC stesso (Data Aggregation, Data Forwarding, Data Communications, Data Validation), nonché applicazioni di controllo/automazione della power grid.

In base alla dimensione di quest'ultima, i dispositivi PDC sono organizzati secondo layer gerarchici: PDC locali, PDC corporate, PDC regionali. Generalmente, i PDC locali aggregano e allineano temporalmente i sincrofasori provenienti dalle PMU, inviando i propri data-stream ai PDC corporate che possono effettuare un controllo qualitativo dei dati ricevuti, quindi li inviano ai PDC regionali che inoltrano il data-stream complessivo alle applicazioni suddette.

[2] https://energycue.it/pmu-misure-diventano-smart/12391/

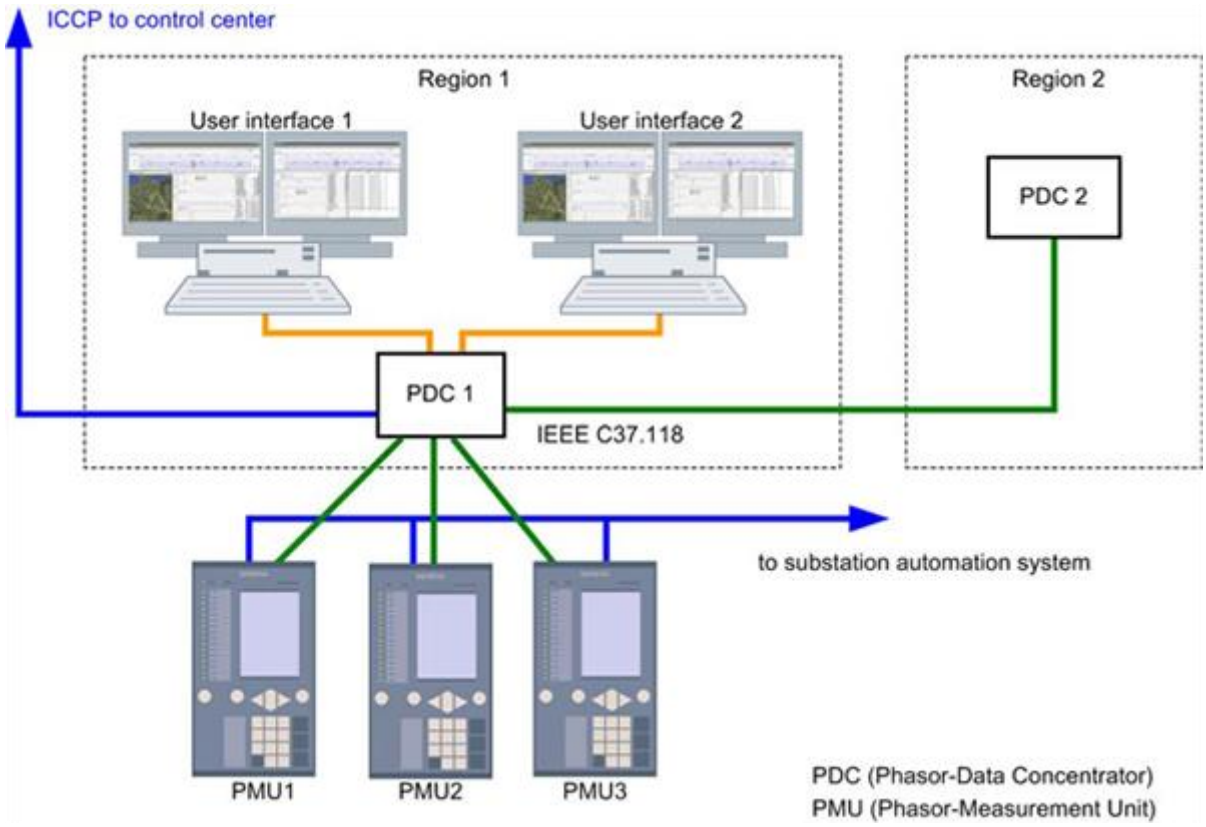

FIGURA 4 – IMPIEGO DEI COMPONENTI FUNZIONALI PDC.

Infine, è da notare che il PDC ha anche la capacità di rilevare eventuali errori semantici a livello applicazione. Ad esempio, sulla base dei valori assunti dai bit 14-15 nel campo STAT del Syncrophasor Data Frame (SDF), esso desume che la PMU inviante tale frame è o in test mode (es., inizializzazione, allineamento, ecc.), oppure non ha inviato alcun SDF (es., PMU down), per cui deve interpretare come non validi i dati ivi presenti oppure contrassegnare come non valido il SDF inviato al PDC di livello superiore in cui, allo scadere di un wait timer, ha inserito una opportuna sequenza di padding. Questa funzione è importante per consentire il riconoscimento in tempo reale di eventuali anomalie funzionali del software di misura.

4.2 Caso d'uso #1: Osservabilità della rete

Un primo caso applicativo che presenta caratteristiche tali da poter potenzialmente beneficiare dei paradigmi Fog/Edge Computing è quello dell'*osservabilità della power grid*.

Nella rete elettrica vengono misurate in maniera sincronizzata una serie di metriche, quali sincrofasori, frequenze, e rate di variazione delle frequenze tramite l'utilizzo di PMU ubicate in corrispondenza dei vari nodi che compongono la rete stessa. L'aggregato di queste misure, relative

a tutti i nodi della power grid, costituisce lo stato della stessa. Nello specifico, l'aggregazione viene effettuata mediante l'impiego di PDC. Secondo quanto già accennato, i dati raccolti da ciascun PDC possono essere ulteriormente aggregati secondo una struttura gerarchica a più livelli (Figura 5). Gli stream di output vengono generati utilizzando i protocolli di comunicazione descritti nello standard IEEE C37.118.2-2011, oppure nello standard IEC 61850-90-5.

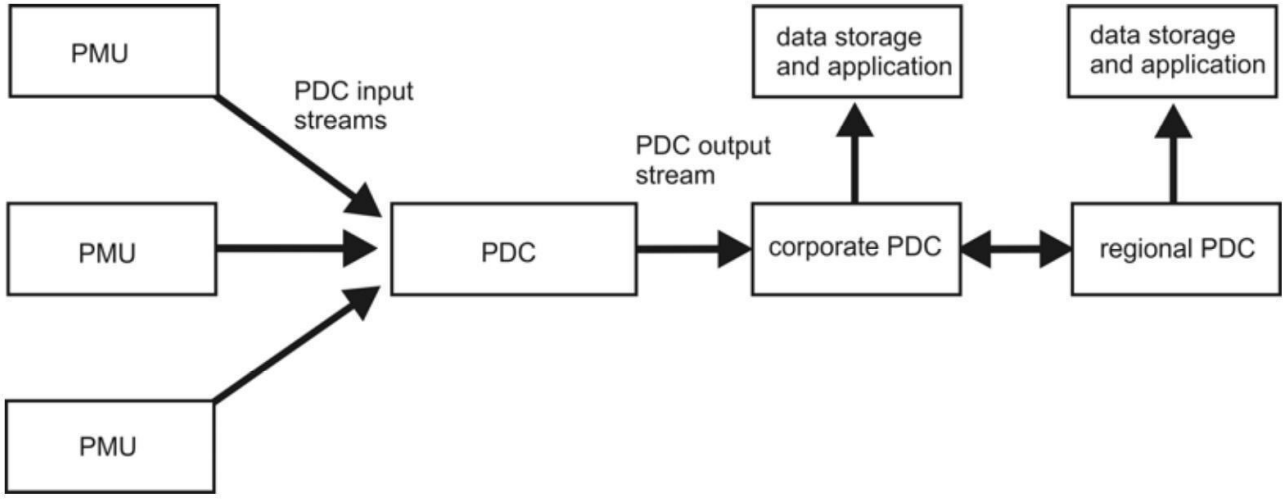

FIGURA 5 – ESEMPIO DI AGGREGAZIONE DELLE MISURAZIONI CON ARCHITETTURA GERARCHICA.

I componenti funzionali PMU/PDC sono fortemente dipendenti dal tipo di hardware/software di supporto al computing; in questo scenario, le tecnologie Fog/Edge Computing possono essere utilizzate per abilitare la migrazione dalle infrastrutture ICT di controllo/automazione attualmente esistenti verso un paradigma *liquido*, in cui il focus è posto non più sull'hardware, bensì sui dati (in tal caso, misure) che le applicazioni mission-critical PMU/PDC devono generare e processare.

Il caso applicativo preso in considerazione viene analizzato facendo riferimento alla configurazione descritta dalla tabella seguente (si distinguono rete di trasmissione / transmission grid e rete di distribuzione / distribution grid, in quanto differenti dal punto di vista topologico).

	Rete di Trasmissione	**Rete di Distribuzione**
Numero di metriche misurate da ogni PMU	30	35
Numero di PMU per nodo	Media di 5 PMU per nodo.	Media di 3-4 PMU per nodo.
Frequenza di aggiornamento misure	20 ms ÷ 1 s	20 ms

Distribuzione geografica	Dispositivi generalmente posizionati nella stazione primaria.	Dispositivi generalmente posizionati nella cabina secondaria.
Livelli di aggregazione di PDC	3 Livelli: stazione primaria, regionale, nazionale.	2 livelli: cabina primaria e concentratore di area.
Sincronizzazione	GPS	GPS

Nell'ottica della caratterizzazione futura delle prestazioni richieste dal caso d'uso e dell'applicabilità definitiva delle soluzioni virtualizzate e orchestrate è ipotizzabile, in futuro, utilizzare il parametro KPI relativo alla **latenza di propagazione dei dati**. Nello specifico, la latenza corrisponde al ritardo di comunicazione complessivo tra una PMU e il PDC di più alto livello considerando tutti gli elementi dell'architettura. Per poter considerare la soluzione fattibile, è necessario che i valori di latenza riscontrati risultino in ogni caso **inferiori ai 100 ms**. Il dettaglio dei contributi di latenza in una architettura con due livelli di PDC gerarchici è mostrato in Figura 6.

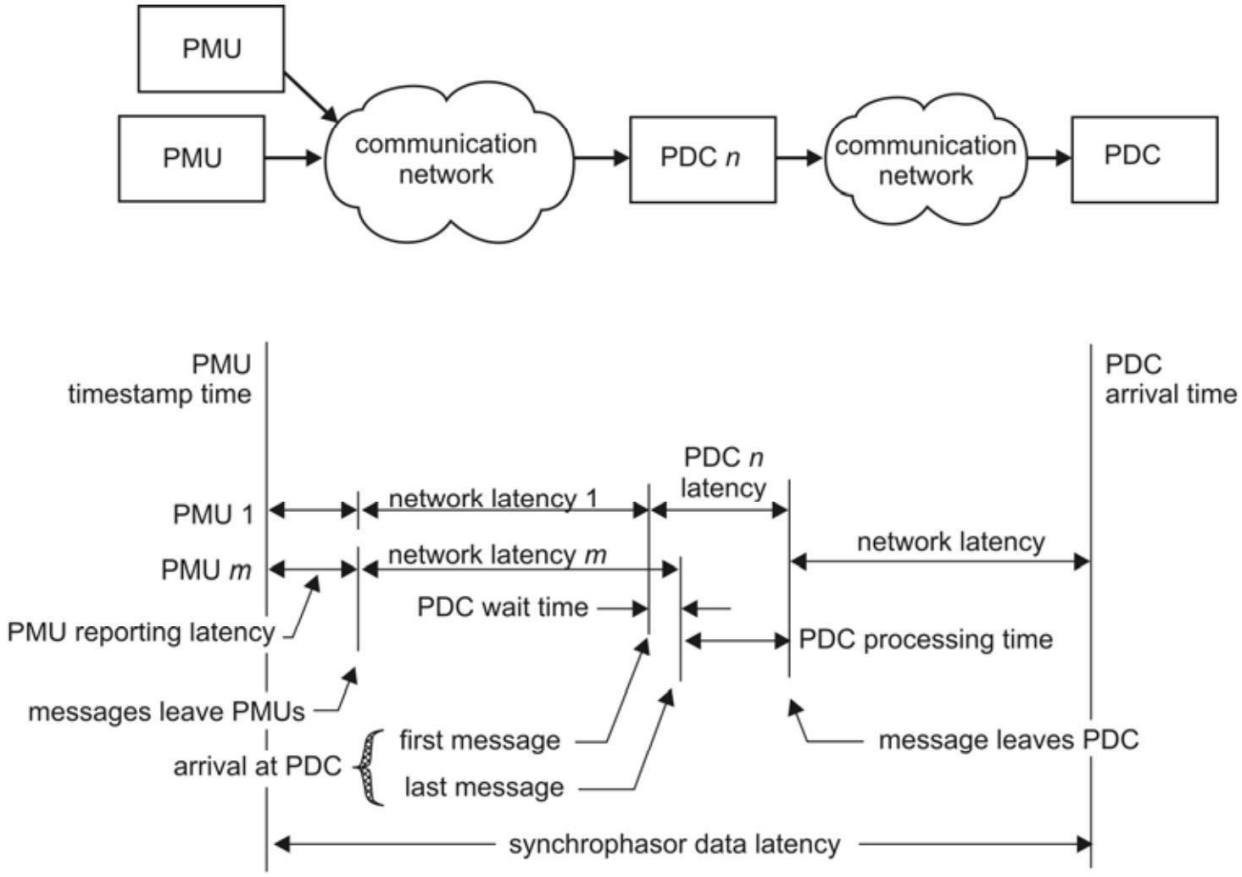

FIGURA 6 – LATENZA DI COMUNICAZIONE TRA UNA PMU E UN PDC DI SECONDO LIVELLO.

4.3 Caso d'uso #2: Controllo di microreti

Nel progettare le future reti di distribuzione assume sempre più rilevanza la necessità di limitare lo spreco di energia elettrica e lo sfruttamento di fonti di energia rinnovabili (vedi "Piano Nazionale Integrato per l'Energia e il Clima" – PNIEC [MSE2020]). Per questo motivo, si prevede che nelle reti di distribuzione di nuova generazione potranno verificarsi pesanti sbilanciamenti tra la domanda e la produzione di energia elettrica, che dovranno essere riequilibrate in maniera dinamica ed intelligente mediante l'impiego di algoritmi di controllo sofisticati ed efficienti. Per poter controllare la domanda di energia, questi algoritmi dovranno ricevere una gran quantità di dati da svariate risorse energetiche, inclusi gli utenti residenziali. Si stima che una abitazione possa generare dai 30 ai 120 dati al secondo [Kansal2012]. A partire dall'aggregato dei dati ricevuti, gli algoritmi di controllo inferiscono le apparecchiature su cui agire e l'insieme di azioni più appropriato da eseguire per sfruttare al meglio le risorse energetiche rinnovabili (ad esempio, vengono comunicati nuovi *set points*). Al fine di ottenere una efficacia ragionevole, l'algoritmo di controllo deve essere in grado di operare in tempo reale: tale necessità di algoritmi sempre più veloci nei tempi di convergenza (*real-time*) comporta il requisito della bassa latenza di propagazione, quindi disponibilità ad essi, dei dati generati dai dispositivi periferici. In altri termini, la propagazione dei dati dai dispositivi periferici al luogo in cui gli algoritmi di controllo sono eseguiti (e viceversa) deve avvenire in tempi rapidissimi.

Il Fog Computing viene visto come una soluzione infrastrutturale abilitante per questo tipo di use case: le tecnologie cloud-like, infatti, consentono flessibilità e scalabilità nella realizzazione ed esecuzione degli algoritmi di controllo, mentre l'architettura distribuita di nodi Fog/Edge risulta essenziale per garantire le latenze richieste da questo caso applicativo. In questo senso, una recente visione è quella che prevede il passaggio dalla Smart Grid tradizionale ad una struttura altamente distribuita, basata su microreti (*micro grid* - µG) [Bernardes2019]. L'idea è che ogni µG abbia la capacità di operare sia quando connessa alla rete di distribuzione nazionale, sia in maniera isolata. Operare in modalità isolata può rendersi necessario in casi di scarsa connettività temporanea o, ad esempio, nei casi in cui bisogna ridurre i tempi di reazione del control-loop tramite funzioni di controllo locali (Figura 7).

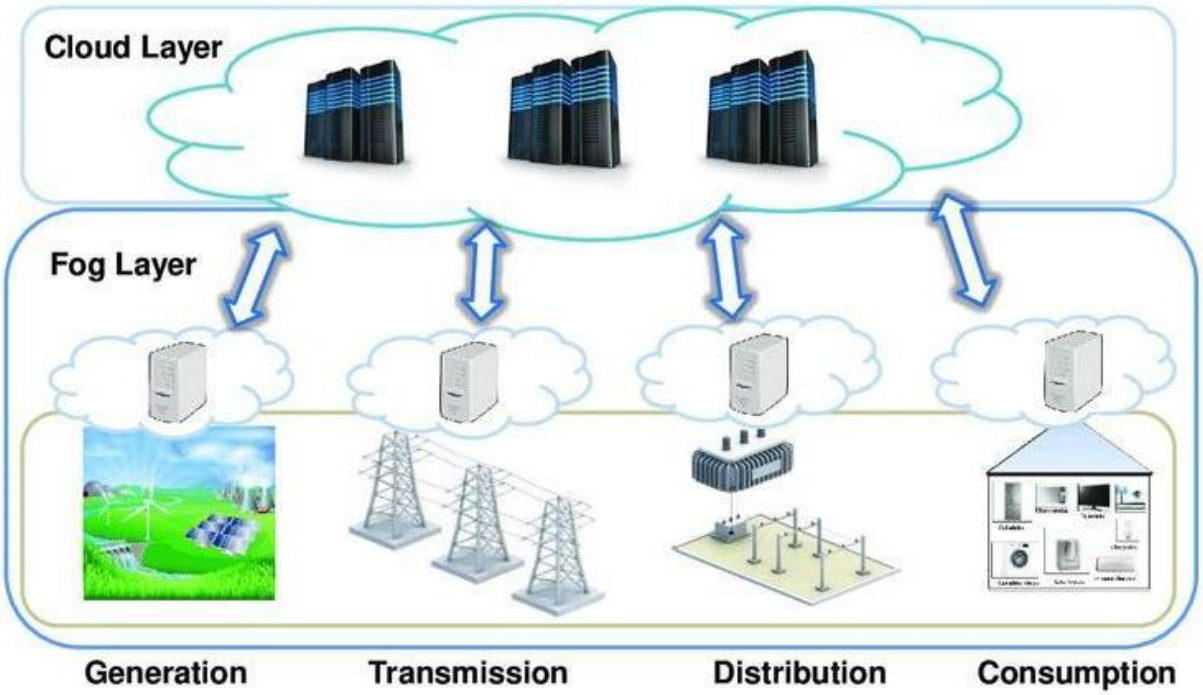

FIGURA 7 – ARCHITETTURA GERARCHICA A MICROGRIDS ABILITATA DA INFRASTRUTTURA FOG/CLOUD.

Il caso tipico che è stato qui preso a riferimento consiste nel controllo di tensione (e della frequenza nel caso di funzionamento in isola) di una microrete con presenza di risorse rinnovabili e flessibili, carichi controllabili ed accumuli. Ogni unità della microrete può essere dotata di una funzione di controllo primaria che, monitorando la rete in tempo reale, può impostare rapidamente (latenza nulla) nuovi *set points* su ogni unità di generazione e carico della microrete, secondo un'azione di controllo locale sulla singola unità. In aggiunta, una funzione di controllo secondaria (es., allocata sui nodi Edge) può agire da coordinatrice e impostare set points più intelligenti sull'intera microrete (avendo a disposizione più informazioni), secondo un'azione di controllo globale sull'insieme di unità di generazione e carico che compongono la microrete. Per l'ottimizzazione economica a lungo termine, una funzione di controllo di terzo livello (ad esempio allocata sul Cloud) può a sua volta coordinare più microreti, influenzando le decisioni dei controllori di secondo livello.

Ai fini dell'applicabilità di tecnologie Fog/Edge, il focus ricade sul controllore di secondo livello. A partire dallo stato della rete di distribuzione acquisito in tempo reale (descritto in sotto-sezione 4.2), durante il funzionamento della microrete sia in connessione alla rete principale (distribuzione) sia in isola, la funzione di controllo dovrà calcolare il set point ottimale per ogni risorsa energetica controllabile disponibile nella microrete e inviare i comandi necessari a mantenere nei limiti desiderati la tensione e la frequenza di esercizio della microrete stessa.

Il caso applicativo preso in considerazione viene analizzato facendo riferimento alla configurazione descritta della tabella seguente.

	Livello primario	**Livello secondario**	**Livello terziario**
Metriche misurate da ogni dispositivo	10	10	10
Numero dispositivi (PMU)	1	100	100
Frequenza di aggiornamento misure	10 ms	1 s	60 s
Periodo del ciclo di controllo	100 ms	1 s ÷ 5 s	60 s
Distribuzione geografica	Singolo dispositivo	Microrete	Remoto (più microreti)
Livelli di aggregazione	Controlla il singolo dispositivo	Controlla più dispositivi sotto più cabine all'interno della stessa microrete	Coordina più microreti a livello nazionale

Nell'ottica della caratterizzazione futura delle prestazioni richieste dal caso d'uso e dell'applicabilità definitiva delle soluzioni virtualizzate e orchestrate è ipotizzabile, in futuro, utilizzare il parametro KPI relativo alla **latenza introdotta dal control-loop**. Nello specifico, la latenza complessiva è data *(i)* dal ritardo di comunicazione (RTT) tra i dispositivi e il server (nodo Fog/Edge) in cui viene eseguita la funzione di controllo e *(ii)* dal tempo necessario alla funzione di controllo per prendere la decisione sui set point ottimali una volta ricevuti i dati. Per poter considerare la soluzione fattibile, è necessario che i valori di latenza riscontrati sul control-loop secondario risultino in ogni caso **inferiori ai 100 ms**. I valori richiesti per ogni livello di controllo sono riportati nella tabella seguente.

	Livello primario	**Livello secondario**	**Livello terziario**
Latenza tollerata	1 ms	100 ms	1 s

5 Gestione di eventi critici: confronto tra le tecnologie disponibili

Partendo dai casi d'uso descritti nel capitolo precedente, questa sezione analizza le caratteristiche delle applicazioni e dei componenti funzionali rispetto alle possibili strategie implementative, ossia rispetto all'implementazione attraverso dispositivi fisici tradizionali, dispositivi virtualizzati con e senza l'introduzione di soluzioni di orchestrazione.

Viene dunque stilata una lista di eventi rilevanti che possono provocare l'interruzione del servizio (es., si verifica che uno dei componenti non è più raggiungibile) e, per ciascuno di tali eventi, vengono individuate le azioni necessarie al ripristino del corretto funzionamento del sistema.

Allo scopo di identificare i potenziali vantaggi dell'orchestrazione, nell'analisi dei suddetti eventi vengono distinti tre casi:

- i componenti in questione sono in esecuzione su dei dispositivi fisici dedicati;
- ciascun componente è in esecuzione all'interno di un ambiente virtualizzato (nello specifico, containerizzato);
- i componenti, oltre ad essere virtualizzati, sono gestiti, configurati e coordinati in maniera automatica da un orchestratore.

5.1 Componente funzionale critico

Ai fini della valutazione degli eventi ICT rilevanti, viene definito il **Componente Funzionale Critico** (**CFC**) nel modo seguente:

- **Fisico (CFC Fisico)**: dispositivo fisico embedded special-purpose / appliance dedicato, attualmente in produzione (es., device National Instruments nel caso PMU) che integra in modo inscindibile la componente hardware e software; in particolare, esegue la funzione mission-critical di misurazione/controllo PMU/PDC come un usuale processo software.
- **Virtualizzato (CFC Virtualizzato)**: funzione software mission-critical di misurazione/controllo PMU/PDC virtualizzata in container Linux (la funzione containerizzata viene eseguita come "main process" del container stesso) in esecuzione su un dispositivo host hardware general-purpose (detto *nodo di computing*). Sotto alcune condizioni, questa soluzione è in grado di disaccoppiare il dispositivo host fisico (hardware general-purpose che offre supporto di computing alla virtualizzazione) dalla funzione software containerizzata che deve essere eseguita, permettendo di rilocare *manualmente* la funzione software su host diversi.
- **Virtualizzato e Orchestrato (CFC Orchestrato)**: funzione software mission-critical di misurazione/controllo PMU/PDC virtualizzata in container Linux (la funzione containerizzata viene eseguita come "main process" del container stesso), poi incapsulato entro un Pod per

essere orchestrato da un orchestratore, che lo pone in esecuzione grazie al supporto di processing offerto da un dispositivo host hardware general-purpose (detto *nodo di orchestrazione*). Sotto alcune condizioni, questa soluzione è in grado di disaccoppiare il dispositivo host fisico (hardware general-purpose che offre supporto di computing a virtualizzazione e orchestrazione) dalla funzione software containerizzata che deve essere eseguita: essendo incapsulata entro un "Pod", il componente di orchestrazione permette di rilocarla *automaticamente* su host diversi, cioè l'orchestratore offre la possibilità di porla automaticamente in esecuzione su un qualunque nodo worker del cluster.

È tuttavia opportuno precisare che il CFC, definito come **funzione software**, può avere delle **dipendenze da eventuali periferiche hardware**. Più in dettaglio, mentre un PDC è un processo puramente software che interagisce con gli altri componenti esclusivamente via rete, una PMU comprende un processo software che interagisce con una periferica fisica (lo strumento hardware di misura). In tal caso, il CFC di una PMU non può essere rilocato su un qualsiasi host a piacere, in quanto il dispositivo che ospita il processo software deve necessariamente essere dotato delle estensioni hardware necessarie all'acquisizione delle misure (sensoristica). Dunque, nel caso del CFC PMU, si inserisce un vincolo sulla flessibilità offerta dalle tecnologie di virtualizzazione, in quanto il dispositivo host fisico che ospita il CFC deve obbligatoriamente essere equipaggiato con lo strumento hardware di misura richiesto, in aggiunta ad avere sufficiente disponibilità di risorse di computazione.

Pertanto, la ridondanza del CFC PMU può essere ottenuta esclusivamente attraverso la ridondanza fisica del dispositivo di acquisizione delle misure, con il vincolo aggiuntivo relativo allo scheduling, consistente nel fatto che il nodo di computing alternativo (dotato di sensoristica) debba trovarsi fisicamente in un intorno geografico opportuno affinché il nuovo componente funzionale (PMU) ivi schedulato acquisisca le misurazioni nel luogo corretto della grid (in pratica, sfruttando la ridondanza di IED/worker node fisicamente adiacente o prossimo).

Contestualizzando questo problema nello scenario elettro-energetico e, particolarmente, considerando lo scenario della *distribution grid*, la non corretta operatività del componente funzionale PMU potrebbe non rappresentare una criticità poiché è ipotizzabile la ridondanza di esso con ulteriori componenti PMU posti nelle vicinanze e collegati alle stesse linee della power grid nelle quali opera, in modo non corretto, quello soggetto al fault. Viceversa, l'impatto di tale evento risulterebbe molto meno trascurabile e potenzialmente critico se il componente funzionale PMU si assuma posizionato nel contesto della *transmission grid*, in tal caso mission-critical in quanto è ipotizzabile (al peggio) non vi sia disponibilità di ulteriori componenti funzionali PMU alternativi interconnessi alle linee di potenza della transmission grid in prossimità di quello guasto e che possano quindi offrire la continuità di servizio.

Alla luce di tali considerazioni si può desumere che, in generale, l'obiettivo di ottenere un buon livello resilienza del monitoraggio delle grandezze elettro-energetiche e del controllo della Smart

Grid richiede un deployment massivo (su larga scala) di dispositivi embedded con le caratteristiche computazionali e fisiche (es., IED/worker node più sensoristica/schede di acquisizione misure) tali da permettere la ridondanza del monitoraggio e del controllo. In altre parole, **le tecnologie di virtualizzazione e orchestrazione, da sole, non possono garantire i risultati voluti se non supportati dalla disponibilità di dispositivi fisici in grado di consentire l'adeguata resilienza/ridondanza**.

Infine, si menziona qui una caratteristica importante delle soluzioni basate su **CFC Fisico**: **la grande maggioranza delle operazioni da effettuare su questo dispositivo** (reboot, manutenzione, aggiornamento) **richiede la presenza in loco di personale specializzato** che interagisca fisicamente con il dispositivo, con un conseguente impatto significativo nell'organizzazione del lavoro e nelle competenze richieste alle figure tecniche che operano sul campo. Invece, nel caso di soluzioni virtualizzate/orchestrate, parte di queste operazioni possono essere condotte e completate da remoto o con limitato supporto locale (es., operazione di attivazione e collegamento alla rete di un nuovo nodo di computing da attuarsi in loco, mentre configurazione successiva condotta o da un operatore remoto oppure da un orchestratore).

5.2 Evento #1. Il CFC ha un guasto hw/sw critico

Viene qui preso in considerazione il caso in cui il CFC Fisico smetta di operare (es., a causa di crash del SO embedded a bordo del device, di malfunzionamento hardware o di alimentazione, ecc.). In tal caso è necessario rendere nuovamente operativo l'intero componente funzionale, considerato come unicum inscindibile di embedded device special-purpose in cui è integrato il software (specialmente nel caso di PMU). Ciò può essere fatto, ad esempio, effettuando manualmente il troubleshooting che porti a stabilire la causa origine del down-time e, quindi, a ripristinare (es., forzando il reboot del dispositivo) o sostituire in toto (sempre manualmente) il CFC Fisico stesso.

Nel caso di CFC Virtualizzato, esso, ad esempio, può aver subìto un crash del proprio "main process" interno (software PMU/PDC), per cui il suo ripristino può avvenire più agilmente ricreandolo e allocandolo manualmente sullo stesso dispositivo host di supporto alla virtualizzazione (es., sullo stesso nodo di computing).

Nel caso di CFC Orchestrato, il container può aver subìto un crash del proprio "main process" interno (software PMU/PDC), per cui il ripristino operativo del componente funzionale (inteso come Pod incapsulante il container con il software PMU/PDC) è effettuato automaticamente dall'orchestratore che, senza alcun intervento umano, provvede a ricrearlo, schedularlo e allocarlo o sullo stesso nodo di computing di orchestrazione, oppure su un altro eventualmente disponibile alla schedulazione (in base a metriche proprie dell'orchestratore stesso, ad esempio relative alle risorse CPU/RAM libere), fatto salvo il limite imposto allo scheduling dall'eventuale dipendenza del CFC da periferiche hardware (es., PMU).

La seguente tabella schematizza la gestione dell'evento in questione, identificando vantaggi e svantaggi delle principali soluzioni a cui si è fatto riferimento nella Sezione 3.

	Fisico	Virtualizzato	Orchestrato
Impatto del guasto	▪ *Irreversibile*, necessario un intervento umano manuale in loco per il ripristino.	▪ *Reversibile* ma è comunque necessario un intervento umano (da remoto) per il reboot sul nodo di computing.	▪ *Reversibile*, non è necessario alcun intervento umano per reboot e re-allocazione delle risorse.
PROs	▪ Semplicità di ripristino nel caso di sostituzione del device hardware.	▪ La funzione containerizzata può essere ricreata manualmente da remoto.	▪ L'orchestratore rialloca ed esegue automaticamente la funzione containerizzata dove più opportuno. ▪ Reazione immediata.
CONs	▪ Reattività imposta da latenze umane. ▪ Necessario un operatore sul campo per il ripristino.	▪ Reattività imposta da latenze umane.	

5.3 Evento #2. Il CFC non è più raggiungibile

Si considera il caso in cui il CFC Fisico sia correttamente operante ma, per un problema di rete esso non sia più raggiungibile attraverso la rete dati della SG. Non conoscendo a priori la causa dell'interruzione del collegamento, è necessario avviare procedure di ricerca guasti sulla rete dati (switch, router, ecc.) che interconnette il dispositivo, in tutta la tratta fino al dispositivo stesso.

Questo evento ha una gestione sostanzialmente simile al caso precedente, con i relativi vantaggi e svantaggi. Dal punto di vista dell'infrastruttura di rete, sia nel caso di tecnologie tradizionali (routing) che nel caso di SDN, è possibile verificare l'esistenza di un percorso secondario per raggiungere il CFC.

I vantaggi e gli svantaggi delle principali soluzioni relativi a tale evento sono schematizzati nella tabella precedente.

5.4 Evento #3. Le risorse hardware non sono più sufficienti per il CFC

Questo evento si verifica in due casi:

1. quando il componente funzionale (es., la funzione software mission-critical di misurazione/controllo PMU/PDC) in esecuzione nel device embedded special-purpose richiede una quantità di risorse hardware (es., CPU, RAM, banda di rete) maggiore rispetto ai valori nominali stimati mediamente per la sua esecuzione. Ciò può avvenire per cause non ordinarie che durante il normale esercizio non dovrebbero verificarsi (es., bug release software, errato dimensionamento / sottodimensionamento delle risorse by design, ecc.).
2. Quando una funzione ausiliaria, anche non strettamente inerente al dominio elettro-energetico, implementata per compiere azioni accessorie (es., management remoto, monitoring, logging, ecc.) ed eseguita sullo stesso device host ove esegue la funzione software mission-critical di misurazione/controllo PMU/PDC, tende a saturare le risorse disponibili sul sistema host[3]. È opportuno precisare che l'evento in questione non può verificarsi se l'hardware del sistema host che supporta il processing è costituito da un device embedded special-purpose/appliance dedicato in cui l'unico processo in esecuzione è il software mission-critical di misurazione/controllo PMU/PDC. Tuttavia, questo caso si rivela sempre meno comune in quanto tipicamente anche i dispositivi embedded sono ormai gestiti da distribuzioni Linux ridotte che, pertanto, espongono il device (quindi l'esecuzione del componente funzionale) alle vulnerabilità di quel sistema operativo e delle sue applicazioni.

In ambo i casi, ne può conseguire la progressiva richiesta di maggiori risorse hardware (che, al peggio, potrebbe provocarne la saturazione sul nodo), dunque l'esecuzione della funzione critica in condizioni di estremo carico. Nel caso di CFC Fisico il cui processo software richiede maggiori risorse, tale evento non è mitigabile poiché, per sua natura, l'hardware di tale device ("resource-constrained") non consente alcuna flessibilità e adattabilità a richieste di carico maggiori di quelle prestabilite inizialmente (dal progetto stesso dell'appliance dedicato) e previsti per l'esecuzione del software PMU/PDC.

Nel caso di CFC Virtualizzato, se il relativo container richiede una quantità di risorse hardware maggiore rispetto alle quote inizialmente stabilite e garantite al container stesso, le si può variare e far scalare manualmente rispettando il limite massimo imposto dall'hardware sottostante del nodo di computing sul quale esegue il kernel del SO che abilita la virtualizzazione del container. Si tratta cioè di uno scaling manuale condotto compatibilmente con le (e limitatamente alle) risorse disponibili sul nodo di computing che ospita l'esecuzione del CFC Virtualizzato.

È tuttavia da notare come nel caso di virtualizzazione leggera la probabilità di riscontrare un peggioramento del servizio critico, dovuto a un consumo anomalo di risorse da parte di servizi

[3] Recentemente si sono verificati numerosi casi in cui nei dispositivi embedded sono stati trovati software inattesi (es., "cryptomining", operanti per creare nuovo denaro virtuale), installati abusivamente da terzi sfruttando note debolezze dei sistemi ("*QNAP warns users of a new crypto-miner named Dovecat infecting their devices*", 21 gennaio 2021, https://zd.net/39W9CWm).

accessori, è minore rispetto allo scenario precedente. Infatti, seppur le primitive kernel di virtualizzazione leggera consentano di limitare, isolare o negare opportunamente l'utilizzo delle risorse hardware a funzioni o gruppi di funzioni software rispetto a quelle disponibili nella virtualizzazione "full", offrono comunque migliori garanzie di isolamento di quanto disponibile nel caso di "bare hardware", che deve essere gestito manualmente (es. con la configurazione esplicita di Linux *cgroups* e *namespaces*).

Nel caso di CFC Orchestrato, lo scaling delle risorse computazionali è demandato all'orchestratore che, secondo metriche/limiti di risorse preimpostate da management del cluster di nodi di computing, in modo automatico (senza alcun intervento umano) provvede reattivamente ad *assegnare* (secondo il c.d. "*vertical Pod autoscaling*" nella terminologia K8s) nuove risorse al componente funzionale (Pod con container del software PMU/PDC), oppure a *spostarlo* su un nodo di computing adiacente che possa offrire maggiori risorse, oppure ancora *replicarlo* in molteplici istanze ("*horizontal Pod autoscaling*"), sullo stesso nodo di computing o su altro nodo sulla base dell'attuale disponibilità di risorse, in modo da garantire le performance computazionali necessarie al componente funzionale stesso.

In altre parole, l'orchestratore può sia assegnare più risorse al servizio in esame qualora queste siano disponibili sul nodo stesso (eventualmente diminuendo le risorse assegnate ad altri servizi di "disturbo") oppure, se l'hardware sottostante al nodo di computing corrente risultasse non sufficiente, l'orchestratore potrebbe avviare nuove istanze del componente funzionale su un nodo di computing adiacente che possa offrire maggiori risorse.

I vantaggi e gli svantaggi delle principali soluzioni relativi a tale evento sono schematizzati nella tabella seguente.

	Fisico	**Virtualizzato**	**Orchestrato**
Classificazione	▪ *Irreversibile*, necessario un intervento umano manuale in loco/in remoto (secondo il caso, es. saturazione totale e turn-off temporaneo o saturazione parziale e reboot).	▪ *Reversibile*, necessario un intervento umano manuale di verifica in remoto sul nodo di computing per effettuare scale-UP oppure scale-OUT (limitati al singolo nodo).	▪ *Reversibile*, non necessario alcun intervento umano per effettuare scale-UP oppure scale-OUT (sia sul nodo stesso che su altri disponibili nelle vicinanze).
PROs	▪ N.D.	▪ Lo scale-UP delle risorse hardware può far fronte alla maggiore richiesta di esse da parte delle funzioni software mission-critical. ▪ Unitamente al vantaggio di cui sopra, le primitive kernel di *light-virt*	▪ Stessi vantaggi già citati per il caso virtualizzato, più i seguenti. ▪ Inspection (monitoring continuo automatico di resource usage). ▪ Automazione completa dei task di "scaling"

		consentono di limitare, isolare o negare opportunamente l'utilizzo delle risorse hardware a funzioni o gruppi di funzioni software.	(provisioning, allocazione ed esecuzione su computing node più opportuno – "controllo adattivo"). ▪ Reattività secondo time-scale (latenze) computing.
CONs	▪ Monitoring delle risorse hardware difficile su embedded device (es., saturazione che impedisce connettività remota). ▪ Il device embedded PMU/PDC non consente scaling delle risorse hardware per sua costruzione (limitatezza intrinseca risorse hardware). ▪ Reattività secondo time-scale (latenze) umane.	▪ Scale-UP risorse hardware virtuali non sempre agevole (es., per mancanza di adeguatezza dell'infrastruttura di computing allo scaling di esse). ▪ Allocazione risorse su nodo di computing non adattiva, per es. secondo criteri implementati da operatore umano (non ottimi perché non sempre basati su metriche precise). ▪ Necessità di un sistema di monitoraggio ad-hoc del CFC, in quanto non disponibile built-in nella piattaforma di virtualizzazione. ▪ In caso di virtualizzazione leggera (es., containers), l'isolamento delle risorse computazionali è più "lasco" poiché non garantito da primitive implementate in HW (contrariamente alla *full virtualization*). ▪ Reattività imposta da latenze umane. ▪ Necessità di hardware fisico abilitante lo "scaling".	▪ Ulteriori componenti e maggiore complessità di configurazione di essi da gestire (es. componente metrics-server, configurazioni di resource quota). ▪ Necessità di hardware fisico abilitante lo "scaling".

5.5 Evento #4. Il CFC è operativo, ma si comporta in modo anomalo

Questo evento considera il caso in cui il componente funzionale PMU in esecuzione sul device embedded special-purpose riporta un errore di funzionamento dell'hardware sottostante (es., fault

ai sensori on-board e/o all'hardware di processing/microcontrollori[4] che non consentano di produrre misure integre delle metriche di interesse). Tale evento è accidentale/asincrono (es., è un guasto hardware, un crash della funzione software, ecc. fin qui visti) ma, rispetto agli altri, presenta la peculiarità di implicare la continuità operativa del componente funzionale, ossia il componente è raggiungibile via rete e potenzialmente genera un flusso dati apparentemente valido.

Infatti, a differenza di tutti gli altri eventi discussi e analizzati in precedenza, quello qui analizzato non causa l'interruzione della funzionalità operativa (es., esecuzione) del componente funzionale PMU - cioè non causa un'interruzione del flusso dati di misura (es., sincrofasori) determinata invece da un possibile crash del componente - che continua a essere operativo, rendendo tale evento molto più subdolo rispetto agli altri eventi asincroni esaminati in quanto non rilevabile semplicemente monitorando il numero di bytes in uscita dal dispositivo (si veda il caso del PDC, sotto-sezione 4.1.2).

Un possibile meccanismo di rilevazione di tale anomalia richiede importanti personalizzazioni delle implementazioni delle tecnologie di virtualizzazione e, soprattutto, di orchestrazione, in quanto è necessario entrare nella semantica del protocollo applicativo e rilevare un'irregolarità dei dati trasmessi. Concettualmente, il problema è analogo alla verifica di operatività di un server web: un meccanismo protocol-agnostic (es., un testing basato sulla possibilità di aprire una nuova sessione TCP) permette di rilevare il fatto che il web server sia ancora attivo (cioè risponda alle richieste di apertura di connessioni TCP) ma non consente di desumere se il server web stia restituendo un errore (es., "HTTP 500 – Server error" anziché "HTTP 200 – OK"). Per la verifica del secondo aspetto, si rende necessaria una sonda in grado di interpretare i codici di risposta a livello applicativo, ossia una sonda *HTTP-aware*.

In presenza di un simile meccanismo di rilevazione, ad esempio mediante tecnica di *snooping* a livello applicativo sugli SDF stessi oppure integrata all'interno del container PDC, eventuali alert così generati potrebbero essere da esso inviati a un control center remoto in cui si provveda manualmente a istanziare un altro container PMU su un dispositivo host di computing adiacente (alternativo a quello soggetto ai possibili fault hardware accennati sopra), quindi ridondato, che disponga di sensoristica connessa alle stesse linee/nodi di potenza della SG.

È dunque evidente come i comportamenti operativi dei componenti funzionali PMU appena analizzati implichino complessità implementativa molto elevata sia sul piano della virtualizzazione, sia sul piano dell'orchestrazione. Infatti, l'analisi compiuta conduce alla considerazione paradossale secondo cui potrebbe essere meglio se il componente funzionale PMU subisse il crash (interrompendo la propria esecuzione) rispetto al caso in cui invii al componente funzionale PDC synchrophasor data-stream marcati come non validi (es., SDF marcati sul bit 14 del campo STAT che

[4] L'errore/guasto potrebbe avvenire nella parte di processing (es., sui microcontrollori o periferici) a bordo dell'IED/Raspberry-Pi e/o nella sua parte "passiva" di sensoristica off-board (es., sul sensore di tensione connesso al bus GPIO del Raspberry-Pi).

indichino malfunzionamento), in quanto il flusso dati è comunque presente e un orchestratore *vanilla* (generico, non opportunamente customizzato secondo le complesse feature ipotizzate in precedenza) non rileverebbe alcuna anomalia. Pertanto, si dovrebbe agire su un'importante customizzazione sia del software che implementa il componente funzionale PDC virtualizzato che riceve i dati di misura dal componente funzionale PMU (snooping applicativo), sia dell'orchestratore (algoritmo di scheduling).

I vantaggi e gli svantaggi delle principali soluzioni relativi a tale evento sono schematizzati nella tabella seguente.

	Fisico	**Virtualizzato**	**Orchestrato**
Classificazione	▪ *Irreversibile*, necessario un intervento umano manuale in loco.	▪ *Reversibile*, necessario un intervento umano manuale (in loco/in remoto) per ripristino della funzione software (es., PMU) su computing node alternativo. Necessario, in seguito, intervento umano manuale in remoto (o in loco) per verifiche più approfondite su stesso computing node soggetto a malfunzionamento, così da indagarne le cause e risolverlo (troubleshooting approfondito).	▪ *Reversibile*, non necessario alcun intervento umano per ripristino della funzione software (es., PMU) su computing node alternativo. Necessario, in seguito, intervento umano manuale in remoto (o in loco) per verifiche più approfondite su stesso computing node soggetto a malfunzionamento per indagarne le cause e risolverlo (troubleshooting approfondito).
PROs	▪ Semplicità di ripristino nella sostituzione del device hardware/IED.	▪ La funzione software (es., PMU) viene manualmente eseguita (provisioning, scheduling ed esecuzione) sul computing node/device IoT scelto dall'operatore umano come alternativo a quello soggetto a fault (agilità di risposta al guasto).	▪ La funzione software (es., PMU) viene automaticamente eseguita (provisioning, scheduling ed esecuzione) sul computing node più opportuno – "controllo adattivo" (velocità di risposta automatica al guasto). ▪ Automazione completa dei task di rescheduling (provisioning, allocazione ed esecuzione su computing node più opportuno – "controllo adattivo"). ▪ Reattività secondo time-scale (latenze) computing.

CONs	▪ Un device ridondato deve essere manualmente avviato dall'operatore ASAP affinché un componente funzionale alternativo a quello guasto produca nuovamente dati utili. ▪ Monitoring hardware difficile su embedded device. ▪ Reattività secondo time-scale (latenze) umane.	▪ Necessario implementare a livello applicativo/container il monitoring continuo automatico di health check del componente funzionale (controllo dell'applicazione a livello semantico). ▪ Stop manuale della VM (*full-virt*) o kill del container (*light-virt*) nel computing node/device IoT soggetto a malfunzionamento e, successivamente, reboot della VM o respawn del container manuali sul computing node/device IoT alternativo. ▪ Allocazione risorse su nodo di computing non adattiva.	▪ Necessario implementare a livello applicativo/container il monitoring continuo automatico di health check del componente funzionale (controllo dell'applicazione a livello semantico).

5.6 Evento #5. Il CFC (o altri componenti critici) sono soggetti a cyber attack

Nel caso in cui siano presenti vulnerabilità di sicurezza nel software in esecuzione a bordo del device host (ad esempio a livello di SO o sull'attuale versione software del componente funzionale costituito dall'applicazione software di misurazione/controllo PMU/PDC), è possibile che queste vengano sfruttate da soggetti malevoli per provocare DoS (Denial of Service) o, addirittura, iniettare del codice malevolo. In tal caso, è da notare che il rischio della presenza di una vulnerabilità critica aumenta con l'aumentare degli strati software in esecuzione.

Va da sé che la soluzione CFC Fisico, in cui il componente funzionale PMU/PDC viene eseguito su hardware dedicato (device embedded special-purpose), espone una superficie d'attacco ridotta rispetto alla soluzione CFC Virtualizzato che prevede strati di virtualizzazione (es., VM hypervisor, Container Runtime Engine) ed eventualmente ulteriore software necessario alla configurazione e gestione remota (es., Ansible, REST APIs). Nel caso di soluzione orchestrata (CFC Orchestrato), questa superficie d'attacco aumenta ulteriormente, essendo presente il software accessorio necessario alla configurazione, coordinamento e management automatici del cluster orchestrato (es., compute agent di Kubernetes).

Di contro, considerando l'eventualità in cui la vulnerabilità software si trovi sulla funzione software mission-critical (es., software PMU/PDC), una soluzione virtualizzata risulta essere molto più flessibile rispetto a quella fisica, consentendo frequenti aggiornamenti di sicurezza (anziché intervenire sul dispositivo stesso con un "reflashing" dell'intera immagine software). In questo senso i vantaggi aumentano nel caso di orchestrazione che, oltre ad automatizzare gli

aggiornamenti, permette di reagire in maniera più efficace alle criticità isolando il componente funzionale compromesso (Pod che incapsula il container che virtualizza il software PMU/PDC) e rimpiazzandolo con allocazione di nuovi componenti o redirezione del carico verso altri componenti eventualmente già presenti. Bisogna inoltre tenere in considerazione che è molto più probabile che eventuali vulnerabilità di sicurezza siano presenti sui componenti funzionali, piuttosto che sugli strati di sistema, di virtualizzazione e di orchestrazione: i primi sono solitamente più soggetti a modifiche/customizzazioni e di gran lunga meno testati rispetto ai Sistemi Operativi o ai Software mainstream di virtualizzazione e orchestrazione rilasciati dai big cloud provider o sviluppati da comunità molto estese di contributori al software open-source.

Infine, nel caso di soluzione orchestrata, è da notare come un attacco al componente "master" dell'orchestratore potrebbe non provocare, sotto alcune condizioni, nessun disservizio ai nodi "worker" sui quali eseguono i CFC (es., PMU/PDC). Infatti, un attacco di tipo DDoS che metta fuori uso il master renderà impossibile reagire a dei cambi di stato del sistema (istanziare una nuova funzione software, migrare un componente su un altro nodo) ma non impedirà al sistema di funzionare nello stato corrente.

I vantaggi e gli svantaggi delle principali soluzioni relativi a tale evento sono schematizzati nella tabella seguente.

	Fisico	**Virtualizzato**	**Orchestrato**
Classificazione	▪ *Irreversibile*, necessario un intervento umano manuale in loco / in remoto per isolare e contenere il cyber attack (switch-off / disconnessione dalla data network).	▪ *Reversibile*, necessario un intervento umano manuale in remoto (o in loco) per verifiche/reboot su stesso computing node.	▪ *Reversibile*, non necessario alcun intervento umano per attivare eventuali task di security policy enforcement e software security update.
PROs	▪ Attuale familiarità tecnica sull'hardware installato on-field.	▪ Migliore sicurezza VM (*full-virt*), essendo limitata l'attack surface dell'hypervisor (kernel *stripped-down* minimale e ben hardenizzato). ▪ Semplicità e agilità dei task manuali relativi al kill+respawn del container (*light-virt*) compromesso.	▪ Inspection (monitoring continuo automatico di health check). ▪ Automazione completa dei task di security policy enforcement (es., kill container compromesso + respawn su computing node più opportuno in attesa di rimedio al componente funzionale app. software). ▪ Eventuale redirezione dei flussi dati su un diverso host che ospita una

			funzione equivalente a quella compromessa ma correttamente funzionante. ▪ Reattività immediata.
CONs	▪ Monitoring cyber attack non sempre agevole su embedded device. ▪ Ripristino presuppone switch-off / disconnessione del componente funzionale dalla data network (mitigazione manuale fino al rimedio). ▪ Reattività imposta da latenze umane.	▪ Mitigazione manuale fino al rimedio definitivo. ▪ Necessario gestire la sicurezza infrastrutturale (la virtualizzazione aggiunge un ulteriore strato software che aumenta l'attack surface).	▪ Complessità di configurazione (ambiente computing, networking/CNI, security policy, persistenza). ▪ Necessario gestire la sicurezza infrastrutturale (l'orchestratore e il virtualizzatore aggiungono un ulteriore strato software che aumenta l'attack surface).

5.7 Evento #6. Il dispositivo host e/o il CFC necessitano di manutenzione

Tale evento considera il caso in cui il CFC (indifferentemente considerato come host oppure come funzione software) necessiti interventi di manutenzione, ad esempio una reinstallazione del software, con tempi di inattività potenzialmente lunghi.

A differenza degli eventi discussi in precedenza, questo non è accidentale ma rappresenta un evento programmato in un determinato periodo. Tuttavia, benché tale eventualità di manutenzione sia programmata, nel caso in cui il CFC richieda dell'hardware speciale (es., sensoristica per PMU), se non vi è ridondanza (*High Availability*) che offra disponibilità di un ulteriore CFC Fisico, l'interruzione del servizio non può essere evitata.

Nel caso di CFC Virtualizzato si hanno vincoli meno stringenti e più facilmente rilassabili rispetto allo scenario di CFC Fisico. Infatti, in virtù della disponibilità di numerosi dispositivi di computing general-purpose nell'infrastruttura di virtualizzazione clusterizzata e della possibilità di schedulare manualmente ovunque, secondo tempi di reazione umani (possibilmente ridotti), un nuovo container che esegua una nuova istanza "parallela" del componente funzionale sorgente (PMU) e/o destinazione (PDC) dei sincrofasori, i periodi di down-time ("*synchrophasor data-stream disruption*") possono essere minimizzati, al meglio annullati se condotti secondo workflow e tempistiche adeguate (es., start di un nuovo container basato su un'immagine aggiornata contenente il software PMU/PDC aggiornato che preceda l'interruzione/kill del container non aggiornato).

Nel caso di CFC Orchestrato, tale evento implica la necessità di considerare, oltre a un aggiornamento di system software del computing node e/o patch della funzione software mission-critical PMU/PDC containerizzata e incapsulata entro un Pod, anche la possibilità di eseguire

l'eventuale update dello strumento di orchestrazione, cioè dell'intera infrastruttura di orchestrazione.

Per evitare down-time, nel caso di installazione patch di system software su IED/edge computing node (es., SO Raspbian on-board al Raspberry-Pi) da aggiornare, si può sfruttare il supporto nativo dell'orchestratore alla gestione di down-time dei propri computing node (visibili come IED/edge worker node): l'orchestratore è capace di migrare i container/Pod della funzione mission-critical PMU/PDC dall'Edge computing node (soggetto all'upgrade del tipo prima detto) su altro worker node del cluster disponibile sul quale eseguirli.

Nel caso di aggiornamento di un'istanza della funzione software mission-critical PMU/PDC containerizzata e incapsulata in Pod, l'orchestratore evita down-time di servizio in maniera trasparente, automatica e a velocità controllata affiancandole una nuova istanza già aggiornata alla quale indirizzare progressivamente i data-stream attuali. Di fatto, l'orchestratore implementa il c.d. *rolling-update* applicativo: effettua il redirect progressivo dei dati PMU/PDC su un'ulteriore istanza del componente funzionale (applicazione containerizzata/Pod) creata ad hoc - come già aggiornata alla versione desiderata - prima di terminare l'istanza non aggiornata. Tale workflow si ripete progressivamente sulle altre istanze di deployment da aggiornare, in attesa del completamento del patching di tutte le istanze applicative containerizzate/Pod presenti su tutti i computing node del cluster. In sostanza, l'orchestratore effettua il deployment (schedulazione, creazione ed esecuzione) di una nuova istanza del componente funzionale aggiornato (Pod incapsulante la funzione software mission-critical containerizzata) prima che l'istanza obsoleta venga inattivata per poi essere distrutta, e così per tutte le istanze obsolete secondo un workflow che preveda un pool di nuove istanze aggiornate.

Inoltre, se l'aggiornamento da condurre dovesse riguardare l'infrastruttura di orchestrazione, si evidenzia la possibile criticità dovuta al fatto che non è detto sia sempre fattibile l'aggiornamento *seamless* del software di orchestrazione, cioè non sempre fattibile senza determinare down-time. In generale, l'aggiornamento dell'orchestratore (quand'anche condotto correttamente) *può* implicare down-time, ad esempio sui talk di monitoraggio (sincrofasori): se non si dispone del supporto al *rolling-update infrastrutturale* automatizzato che effettui progressivi aggiornamenti del software di orchestrazione su tutti i nodi di computing del cluster (control-plane e data-plane), in un ordine prestabilito, può essere necessario prevedere e prepararsi a down-time dell'intero cluster. Tale tipo di aggiornamento automatico infrastrutturale non costituisce infatti una feature già integrata *ab origine* nei software di orchestrazione ma, piuttosto, è integrabile sfruttando ulteriori tool di terze parti e il supporto enterprise di piattaforme di hosting che offrano (a pagamento) tale tipo di servizio.

	Fisico	**Virtualizzato**	**Orchestrato**

Classificazione	▪ *Reversibile*, necessario un intervento umano manuale in loco / in remoto per operations (update).	▪ *Reversibile*, necessario un intervento umano manuale in remoto (o in loco) per aggiornamenti su stesso computing node/IED nel caso di aggiornamento della funzione software mission-critical containerizzata (es., PMU/PDC) oppure su computing node/IED ridondato nel caso di patch di sistema operativo su computing node.	▪ *Reversibile*, non necessario alcun intervento umano per aggiornamenti della funzione software mission-critical containerizzata; necessario un intervento umano manuale in remoto (o in loco) nel caso di aggiornamento del sistema operativo e/o software di orchestrazione sull'intero cluster (es., Kubernetes).
PROs	▪ Attuale familiarità tecnica sia su hardware installato on-field, sia sugli strumenti di remote management (update task che, benché mission-critical cioè da condurre con attenzione, è comunque meccanico e ripetitivo). ▪ Semplicità dell'update task dal punto di vista ICT (qualunque tecnico con competenze ICT base e dovuta attenzione riesce a effettuarlo su embedded device, sia in loco sia in remoto).	▪ Per evitare down-time, nel caso di funzione software mission-critical containerizzata (es., PMU/PDC) da aggiornare, le si può affiancare una nuova istanza già aggiornata alla quale indirizzare progressivamente i data-stream attuali. ▪ Per evitare down-time nel caso di patching di system software su IED, all'IED si può affiancare un secondo IED aggiornato sul quale eseguire una nuova istanza della funzione mission-critical alla quale indirizzare progressivamente i data-stream attuali.	▪ Automazione completa dei task di rolling-update applicativo: l'orchestratore evita down-time nel caso di aggiornamento della funzione software mission-critical containerizzata (es., PMU/PDC), affiancandole una nuova istanza già aggiornata alla quale indirizzare progressivamente i data-stream attuali. ▪ Per evitare down-time, nel caso patching di system software su IED/edge computing node, si sfrutta il supporto nativo dell'orchestratore alla gestione di down-time di IED/edge computing node. ▪ Minima latenza tra scheduling e patching, secondo time-scale del computing.
CONs	▪ Aggiornamento non sempre fattibile "a caldo", potrebbe presupporre switch-off / disconnessione data network, ecc. (ossia l'interruzione	▪ Configurazione solo parzialmente (e non sempre) automatizzabile, prona a errori umani. ▪ Aggiornamento problematico del sistema operativo su IED (interruzione servizio	▪ Aggiornamento dell'orchestratore non sempre fattibile senza down-time (possibile per il nodo "master", mentre per i nodi "worker" può verificarsi down-time).

	dell'invio dei sincrofasori). ▪ Aggiornamento problematico, causante interruzione di servizio mission-critical, nel caso non sia prevista ridondanza di dispositivo/HA.	mission- critical) nel caso non sia prevista ridondanza di nodo/HA.	▪ Aggiornamento dell'orchestratore può implicare down-time se non si dispone del supporto ai rolling-update infrastrutturali automatizzati.

6 Proposta architetturale

Come introdotto dal caso d'uso relativo alla "*Osservabilità della rete*" (sotto-sezione 4.2), essa può attuarsi mediante l'acquisizione di misure di diverse grandezze elettriche utilizzando PMU posizionate in corrispondenza di nodi d'interesse della grid: tali misure forniscono lo stato corrente aggiornato della grid e possono essere utilizzate per eseguire funzioni di controllo/automazione, nonché per assumere decisioni circa l'assetto futuro (cioè lo stato voluto) della grid stessa.

Dati i costi dei dispositivi PMU special-purpose, le PMU sono poche e dislocate solo in nodi "nevralgici" della power grid attuale, sicuramente nella sua parte di *trasmissione* ma mai nella sua parte di *distribuzione*: per abilitare un controllo maggiormente raffinato (a granularità fine) della power grid tendente all'evoluzione in SG, è auspicabile aumentare il numero di tali dispositivi operanti in essa, avvicinandoli il più possibile ai clienti finali (fino al "contatore"), utilizzando device hardware general-purpose a basso costo (cost-effective) che supportino il processing di funzionalità software PMU anche nella *distribution grid*, per migliorarne l'osservabilità e abilitare lo sviluppo di ulteriori applicazioni (es., rilevazione e localizzazione guasti) utilizzando la medesima infrastruttura. Concettualmente, si migra così verso uno scenario tipico di infrastrutture FEC, nelle quali elementi fondamentali sono la distribuzione geografica dei dispositivi IoT, deployment large-scale di sensoristica e sensor networks, elevato numero di computing node eterogenei e loro pervasività capillare nella SG a livello Fog/Edge location, *location awareness* di essi e bassa latenza di comunicazione abilitante il deployment di applicazioni *latency-sensitive* (es., applicazioni di misura funzionalmente implementate dalle PMU), interoperabilità e federazione di esse, che supportino *on-line analytics* e interazione con il Cloud Computing (anche *on-premise*).

Nella infrastruttura FEC proposta, la PMU identifica una *funzione software mission-critical di misura* delle suddette grandezze che viene **virtualizzata e orchestrata**, attraverso un container (*light-virtualization*) il cui supporto di computing è offerto da hardware general-purpose a basso costo (es., Raspberry-Pi). Sebbene la funzione PMU possa trarre apparentemente poco vantaggio da una soluzione orchestrata, in quanto vincolata alla disponibilità di hardware di acquisizione dati connesso al dispositivo stesso, è da notare come una soluzione orchestrata non presenti particolari criticità aggiuntive ma, bensì, offra maggiori funzionalità di gestione del ciclo di vita di un container: per esempio, esso è in grado di accorgersi che il software di un Pod (container della funzione PMU) sia andato in *crash* e di farlo ripartire automaticamente. Inoltre, grazie alla disponibilità di versioni ridotte/minimali di Kubernetes (es., la *stripped-down* K3s), anche le esigenze in termini di CPU/RAM sono compatibili con quanto disponibile su un nodo di piccole dimensioni (es., Raspberry-Pi).

Per quanto riguarda il PDC, a maggior ragione viene preferita una soluzione **virtualizzata** (tramite virtualizzazione leggera / *light-virt*) **e orchestrata**, che consente di creare i servizi applicativi operanti su hardware general-purpose (es., NebbioloTech Fog node, "white label" server et similia) utile ad abilitare applicazioni Fog/Edge computing, interconnesso alla data network della SG esattamente come nel caso tradizionale della power grid attuale. Analogamente al caso PMU già introdotto poco

sopra, tale containerizzazione è necessaria all'implementazione dell'orchestrazione che, anche nel caso PDC, potrebbe consentire la creazione di servizi applicativi mission-critical basati sulle funzioni PDC containerizzate estesi su più sub-layer gerarchici di container incapsulati nei Pod in esecuzione sui computing node general-purpose dispiegati sulla grid. Tali container/Pod, esattamente come nel caso PMU, sono programmati in cluster di risorse delle quali l'orchestratore gestisca automaticamente la scalabilità, la disponibilità e l'integrità nel tempo necessarie all'ambiente di produzione della Smart Grid per espletare in modo efficiente e affidabile il controllo per l'esercizio sicuro di essa.

La gerarchia e le interconnessioni PMU-PDC sopra descritte vengono mappate sull'architettura FEC proposta e modellate secondo uno schema logico gerarchico di forma piramidale, di seguito raffigurato (Figura 8).

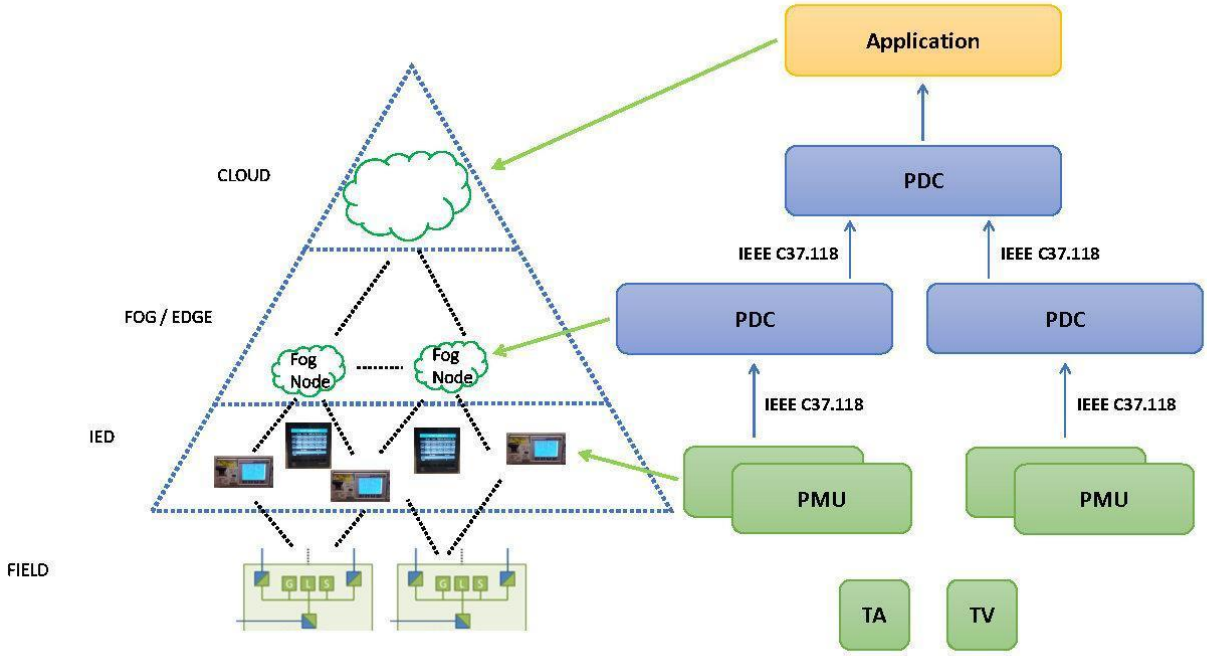

FIGURA 8 – SCHEMA LOGICO GERARCHICO DELL'ARCHITETTURA FEC PROPOSTA.

Al livello base (*Edge/IoT* layer) dell'architettura FEC si trovano i dispositivi IED costituiti da HW general-purpose a basso costo (es., Raspberry-Pi, Arduino, ecc.), che integrano sensoristica on-board connessa alle linee di potenza della power grid e che offrono supporto di computing alle molteplici **funzioni software PMU virtualizzate e orchestrate**, necessarie a generare i sincrofasori inviati ai livelli soprastanti.

Al livello intermedio (*Fog* layer) si trovano i dispositivi HW general-purpose (es., NebbioloTech Fog node, "white label" server et similia) che offrono il supporto di computing alle molteplici **funzioni**

software PDC virtualizzate e orchestrate che ospitano i servizi software caratteristici di un PDC, quali Data Aggregation, Data Forwarding, Data Communications e Data Validation. Come nella power grid tradizionale, per ragioni di High Availability (HA), scalabilità e resilienza del sistema FEC, è necessario prevedere una possibile organizzazione di tale livello intermedio in sub-layer mappati sulla gerarchia PDC (PDC locali, PDC corporate, PDC regionali) aventi stesso ruolo funzionale della power grid attuale.

Al livello superiore/finale (*Cloud* layer), in cima alla piramide architetturale, si trova l'entità di processing necessaria a **orchestrare le funzioni software componenti i due livelli sottostanti**: si tratta del control-plane ospitato in cloud (*on-premise*, nel caso DERTF di RSE S.p.A.) che, logicamente, rappresenta una piattaforma flessibile ed estensibile per l'esecuzione e la gestione elastica e automatizzata di workload computazionali di funzioni software containerizzate. Tale control-plane è responsabile di tutte le decisioni globali sul sistema/infrastruttura FEC (es., scheduling dei task infrastrutturali e delle funzioni software containerizzate di controllo/misura tipiche del dominio elettro-energetico fin qui discusse, rilevazione e risposta a eventi critici di guasto delle componenti hardware e arresto delle funzioni software containerizzate dei livelli sottostanti, avvio di nuove istanze di tali funzioni, ecc.) in modo da soddisfare il raggiungimento dello *stato desiderato/atteso* dell'intero sistema FEC.

7 Implementazione

Tenendo presente l'architettura concettuale e la sua rappresentazione grafica sviluppate nella Sezione 6 precedente, è possibile concretizzarla mediante il setup e il deployment di *test-bed* utili a sperimentare ed esaminare le implementazioni delle tecnologie di virtualizzazione e orchestrazione già descritte in precedenza, contestualizzandole simulando il loro impiego in uno scenario il più possibile aderente ai casi applicativi proposti ("Caso d'uso #1: Osservabilità della rete" e "Caso d'uso #2: Controllo di microreti"), secondo i mezzi a disposizione.

7.1 Containerizzazione dei componenti

Questa sotto-sezione discute l'applicazione pratica della tecnologia prescelta per la containerizzazione: Docker, selezionata per i vantaggi enunciati nella sotto-sezione 3.1.1.1, nonché per la sua attuale compatibilità con Kubernetes e sue varianti.

Nel caso di studio in analisi, le funzioni software mission-critical di misurazione/controllo PMU/PDC che dovranno essere eseguite in Docker container con immagine POSIX (Ubuntu, Raspbian et similia) sono state, in prima battuta, *openPDC* e il software PMU che acquisisce i sincrofasori (i.e. i valori delle metriche del dominio elettro-energetico) ed è responsabile di inviarli alle funzioni containerizzate PDC, ove in seguito verranno processati e inoltrati verso i container PDC di livello superiore.

In riferimento all'obiettivo del progetto, si è provata l'installazione e la successiva esecuzione di openPDC versione 2.1 all'interno di un container Linux tramite la guida dettagliata reperita su GitHub[5]. Per ottenere compatibilità con la versione dei tool ausiliari - quali, ad esempio, *Mono* versione 3.12.1 (cross-platform implementation del framework Microsoft .NET) - necessari per l'esecuzione di openPDC v2.1, è stato creato ed eseguito un container basato sull'immagine del SO GNU/Linux distribuzione Ubuntu 14.04.6 LTS (Trusty Tahr).

Dal momento che sul repository pubblico DockerHub [DockerHub] è disponibile solo la container-image Ubuntu 20.04.1 LTS (Focal Fossa), per creare la container-image Ubuntu 14.04.6 LTS voluta si è sfruttata la feature di *automation build* (integrata in Docker), che consente di creare in maniera automatica una container-image personalizzata partendo dalla specifica dei componenti base di essa (es., specifica della distribuzione del SO, versione kernel, download e aggiunta di package, tool e librerie specifiche, ecc.) e delle azioni da eseguire, tramite la stesura di un *Dockerfile*. Maggiori dettagli sono riportati in "Appendice A: Cenni di configurazione".

[5] openPDC GitHub repository: https://github.com/GridProtectionAlliance/openPDC

Le issue preesistenti e già aperte sul repository GitHub [OpenPDC] mostrano che gli script per l'installazione di *openPDC* v2.1 non sono aggiornati né attualmente manutenuti dallo sviluppatore del progetto, dunque la loro esecuzione riporta esito negativo.

A causa degli errori riscontrati in fase di installazione, la scelta di *openPDC* è stata abbandonata in favore del software di simulazione *iPDC* [iPDC] che, grazie all'interazione con un software di simulazione *PMU* che ne rappresenta la controparte, risulta essere una valida alternativa funzionante. Per realizzare il dispiegamento delle funzioni software di simulazione PMU/PDC appena dette all'interno di Docker container, dapprima sono stati scritti i rispettivi Dockerfile necessari a virtualizzare i suddetti componenti software di interesse, cioè necessari al build delle relative container-images; successivamente, tramite il tool di automazione *Docker Compose*, sono state lanciate (sulla macchina host locale) le esecuzioni dei container istanze di tali immagini che, nello specifico, sono:

- *PMU*, che esegue il software omonimo di simulazione del CFC PMU, cioè l'acquisizione di metriche del dominio elettro-energetico (sincrofasori), creazione e inoltro del proprio synchrophasor data-stream (flusso di SDF) al container *iPDC* (i.e. al CFC PDC containerizzato);
- *iPDC*, che esegue il software omonimo di simulazione del CFC PDC, cioè ricezione dei synchrophasor data-stream dai container *PMU* e inoltro di essi sia al container *DB* per la memorizzazione / persistenza dei dati, sia ai CFC PDC containerizzati di livello superiore;
- *DB*, che costituisce l'interfaccia applicativa tra il software *iPDC* che riceve/aggrega i dati di misura (sincrofasori, stream di SDF) provenienti da un singolo CFC PDC containerizzato (simulato dal container *iPDC*) del livello di appartenenza e il corrispondente DBMS locale necessario a memorizzare tali dati in modo persistente;
- *MySQL*, che rappresenta il DBMS locale (interfacciato con *DB* di cui sopra) necessario a produrre la persistenza - locale alla macchina host - dei dati di misura (sincrofasori, stream SDF) ricevuti e aggregati dai componenti PDC che istanziano il software *iPDC* containerizzato, mediante opportuno database-schema usato in fase di creazione del container stesso.

Create tutte le container-image necessarie, su una macchina host si è realizzata l'infrastruttura di rete virtualizzata nella quale dispiegare i container che istanziano le funzioni software di simulazione dell'acquisizione dei dati di misura e dell'aggregazione/persistenza di essi (*PMU*, *iPDC*, *DB* e *MySQL*). Quindi, si è testata la corretta interazione tra essi mediante Docker Compose, con il quale è stato possibile creare ed eseguire, in locale e in maniera automatizzata, i Docker container istanze delle suddette container-images. Come ci si aspetta da uno scenario reale, come nel caso del componente funzionale PMU fisico attuale, ogni Docker container che virtualizza ed esegue il software PMU è stato dispiegato per operare in una propria sottorete virtuale ad esso dedicata. Come già accennato, tali container PMU simulano l'acquisizione dall'esterno dei valori di misura delle metriche elettriche (sincrofasori), ciascuno originando perciò il proprio synchrophasor data-stream (flusso fi SDF), inviandolo quindi al Docker container che esegue il software *iPDC* (i.e. container del CFC PDC) e agisce da aggregatore degli stream di misure. Quest'ultimo è stato invece creato ed eseguito

insieme ai container *DB* e *MySQL* necessari alla memorizzazione persistente dei dati (ricevuti dal container del CFC PDC) all'interno di una stessa sottorete virtuale, comunque distinta da quelle appositamente realizzate per l'esecuzione dei container PMU. Infine, dato che l'interazione diretta tra i Docker container appartenenti a sottoreti virtuali differenti non è consentita, e volendone comunque garantire un corretto grado di isolamento, se ne è abilitato l'internetworking (comunicazione tra le rispettive sottoreti) attraverso l'interconnesione alla sottorete IPv4 del sistema host fisico sul quale tali container eseguono, abilitando su essa il routing mediante l'implementazione del port mapping (i.e. *port-address translation*). Nella pratica, ricorrendo alla sottorete della macchina host sulla quale eseguono tutti i container visti (i.e. host della *light-virt*), sono state create delle *rules* (regole) responsabili di redirigere il traffico applicativo verso i container dei componenti funzionali simulati PMU/PDC dispiegati nelle rispettive sottoreti di appartenenza, ottenendone dunque la comunicazione attesa.

7.2 Architettura di orchestrazione

A seguito della containerizzazione delle funzioni software mission-critical di misurazione/controllo PMU/PDC, propedeutica all'orchestrazione di esse, in questa sotto-sezione si passano dunque in rassegna le implementazioni delle soluzioni tecnologiche orchestrate inizialmente vagliate nella sotto-sezione 3.2, tra le quali verrà scelta quella che soddisfa criteri di reale ed efficace applicabilità tecnica allo scenario elettro-energetico, nell'ottica della caratterizzazione che verrà affrontata negli sviluppi futuri dell'attività di ricerca.

7.2.1 Implementazione KubeEdge

Il deployment della soluzione di orchestrazione delle applicazioni software/componenti funzionali mission-critical PMU/PDC containerizzate ha visto in prima battuta la scelta di *KubeEdge*, essenzialmente perché presumibilmente esso avrebbe potuto consentire di orchestrare applicazioni, gestire dispositivi IoT e monitorare sia lo stato delle applicazioni sia dei dispositivi sui computing node all'Edge layer proprio come un tradizionale cluster Kubernetes installato e in esecuzione presso il Cloud layer (che sia questo privato, pubblico oppure ibrido), con il supposto vantaggio che il data-plane (agent *EdgeCore* in esecuzione sui nodi worker del cluster sui quali verrebbero schedulati i container/Pod dei software PMU/PDC) supporti la modalità transitoria *off-line*, potendo potenzialmente proseguire l'esecuzione anche se disconnesso dal control-plane (modulo *CloudCore* in esecuzione sul nodo master K8s deployato al Cloud layer) con il quale, alla riconnessione, riconcilierebbe lo stato e sincronizzerebbe i dati applicativi.

L'architettura di KubeEdge teoricamente potrebbe far sì che la logica applicativa (cioè i container PMU/PDC) in esecuzione all'Edge layer produca ed elabori localmente i dati tipici dello scenario d'interesse (synchrophasor data-stream), in volumi cospicui proprio nel luogo in cui vengano generati. Ciò tenderebbe a ridurre la restrittività di eventuali requisiti di throughput disponibile e volumi di dati sulla rete tra Edge layer e Cloud layer, producendo potenzialmente l'effetto di

aumentare la reattività del sistema, proteggere i dati mission-critical e ridurre i costi (nel caso le componenti al Cloud layer siano ospitate presso un cloud provider).

Come accennato in seguito (si veda l'"Appendice A: Cenni di configurazione"), inizialmente si è proceduto alla configurazione della funzionalità *kubectl logs* propedeutica al corretto deployment del componente *metrics-server* (necessario all'auto-scaling applicativo dei container/Pod mission-critical PMU/PDC) che, quando abilitata, in prima battuta non è risultata funzionante. Tale funzionalità è necessaria anche all'ispezione dei log prodotti dai processi applicativi containerizzati interni a ciascun Pod, almeno sotto tale aspetto sfruttata con successo in seguito a ulteriori tentativi di abilitazione di essa andati poi a buon fine. Indipendentemente dalla funzionalità *kubectl logs* prodromica, comunque, il metrics-server previsto dalla soluzione KubeEdge necessita di parecchia customizzazione rispetto alla versione "canonica" presente in Kubernetes vanilla. Bisogna infatti configurarne vari aspetti, quali la generazione di certificati necessari alle comunicazioni sicure, il packet filtering selettivo attuato mediante il tool *iptables*, quindi condurre l'editing massiccio dei file YAML di configurazione dei moduli *CloudCore* ed *EdgeCore*, nonché l'evitare che il *kube-proxy* venga istanziato (o abilitato) su qualunque Edge Worker Node. Questo componente non solo risulta essere particolarmente complesso e articolato nella configurazione ma, purtroppo, la relativa documentazione ufficiale è molto scarna. Per questa ragione il deployment è stato rimandato ad una successiva fase di sviluppo, comunque interrotta sulla base di quanto segue.

Infatti, durante la fase di deployment della soluzione KubeEdge si è evidenziata la criticità dovuta alla scelta del plug-in CNI che potesse essere operativamente adatto e compatibile, dal punto di vista funzionale, con tutti i componenti coinvolti nel deployment stesso: K8s control-plane master node in combinazione a KubeEdge *CloudCore* (entrambi in esecuzione sulla VM KubeEdge Master Node), KubeEdge *EdgeCore* (agent in esecuzione sulla VM KubeEdge Worker Node). La scelta del plug-in CNI, suffragata da alcuni deployment condotti in passato con successo su altri cluster K8s vanilla (quindi per esperienza pregressa positiva) è caduta inizialmente su *Cilium*: Pod network software open-source progettato per implementare la connettività di rete tra i container/Pod e securizzarla in modo trasparente, compatibile con L7/HTTP potendo applicare network policy L3-L7 utilizzando un modello di sicurezza basato sull'identità che è disaccoppiato dall'indirizzamento di rete (supporta la multi-tenancy ed è utilizzabile anche in combinazione con altri plug-in CNI, benché ciò sia sconsigliato a utenti meno esperti). Maggiori dettagli sono riportati nell'"Appendice A: Cenni di configurazione".

Tuttavia, nello scenario attualmente sperimentato, si è evidenziato il comportamento anomalo dei Pod di servizio costituenti il deployment del plug-in CNI che li vede soggetti a "fatal error", stato "Terminating" pendente, a causa dell'inizialmente supposta incompatibilità di Cilium rispetto ai moduli KubeEdge (ne è garantita compatibilità con Kubernetes vanilla). Di fatto, essi risultano soggetti al c.d. *crash loop backoff*, condizione secondo la quale in seguito al comportamento default dell'orchestratore, ad ogni evento di crash i moduli (container interni ai Pod del plug-in CNI) vengono sempre ri-creati, non potendo però espletare le proprie funzioni. Quali possibili alternative

al deployment di Cilium sono stati vagliati, nell'ordine, i plug-in CNI *Flannel* e *Calico*: il primo, in virtù della propria semplicità implementativa interna (crea una rete overlay intra-cluster molto semplice, basata sul protocollo VXLAN, che soddisfa i requisiti di K8s); il secondo, essendo attualmente l'unico plug-in CNI sul quale gli sviluppatori del progetto *kubeadm* (usato per il deployment del control-plane K8s vanilla) eseguono test end-to-end. In seguito ai deployment di questi ultimi (effettuati separatamente l'uno dall'altro, ossia condotti ciascuno su un proprio test-bed KubeEdge differente e dedicato), l'ipotesi iniziale di particolare incompatibilità del solo plug-in CNI *Cilium* con la soluzione KubeEdge non è stata avvalorata, anzi smentita, dall'analisi dei log interni ai Pod di servizio di tali plug-in CNI eseguiti sul KubeEdge Worker Node di ciascun test-bed. Tale analisi, condotta mediante la pre-configurata e attivata funzionalità *kubectl logs*, ha consentito di risalire ad alcune issue reperite (aperte e attualmente non ancora risolte) sul repository GitHub ufficiale di KubeEdge, relative a deployment falliti del plug-in CNI *Calico*: in esse si è rilevato come il progetto KubeEdge non supporti ancora alcun plug-in CNI in quanto lo sviluppo di tale supporto è correntemente argomento di design da parte della comunità degli sviluppatori[6].

Bisogna precisare che il deployment corretto di un plug-in CNI è funzionale per qualsiasi tecnologia di orchestrazione, in quanto indispensabile per realizzare l'infrastruttura software di networking necessaria ai Pod del cluster di orchestrazione mediante la quale essi possano interconnettersi attraverso le primitive dei servizi standard di Kubernetes (es., ClusterIP, NodePort, Load Balancer), esattamente come nel caso dei CFC Orchestrati PMU/PDC. In mancanza di tale supporto al networking, l'orchestratore è sì capace di gestire il ciclo di vita dei container ma non è comunque in grado di fornire le soluzioni di rete proprie di K8s che, ad esempio, permettono a un servizio A di contattare un servizio B attraverso un indirizzo IP che non dipenda dall'attuale server/nodo di computing sul quale quest'ultimo servizio è ospitato. In mancanza di tale caratteristica di rete, per garantire la possibilità ai vari servizi di poter comunicare indipendentemente dalla loro posizione, sarebbe necessario ricorrere a un sistema terzo, ad esempio che implementi un *bus a eventi* basato sul già accennato meccanismo publish/subscribe, quale può essere *Apache Kafka* [ApacheKafka]. Si tratta di una piattaforma distribuita per la gestione real-time di messaggi / eventi scambiati tra endpoint generici, che fondamentalmente implementa le funzionalità logiche di *queueing* (accodamento), *processing* (elaborazione) e *storage* (archiviazione) dei messaggi. Essa si propone l'obiettivo di gestire throughput elevati, tali da supportare flussi di eventi ad altissimo volume (es., aggregando log in real-time) ma, soprattutto, garantire bassa latenza di consegna dei messaggi. Tutte caratteristiche, queste, di sicuro interesse nello scenario elettro-energetico.

Per quanto sopra, la soluzione KubeEdge è stata accantonata - almeno finché la comunità di sviluppo non implementi il supporto sopra descritto - in favore della soluzione *K3s/Rancher*.

[6] https://github.com/kubeedge/kubeedge/issues/2281

7.2.2 Implementazione K3s/Rancher

Come accennato in precedenza, K3s/Rancher costituisce una possibile soluzione al problema dell'interoperabilità tra cluster di orchestrazione diversi che, potenzialmente, potrebbero operare anche in maniera autonoma. Sulla base di tale considerazione, si è proceduto alla realizzazione di un test-bed che, differentemente dal caso KubeEdge, durante la fase di deployment non evidenziasse alcuna criticità primariamente nella funzionalità effettiva del plug-in CNI. Rispetto al caso KubeEdge, K3s/Rancher risulta decisamente più semplice, compatto e agevole dal punto di vista operativo dell'installazione e della configurazione di tutti i componenti coinvolti nel deployment stesso: K3s *Server* Node (control-plane master node) in combinazione al DBMS *SQLite3* che offre persistenza allo stato condiviso del cluster (entrambi in esecuzione sulla VM K3s Master Node), K3s *Agent* Node (data-plane worker node in esecuzione sulla VM K3s Worker Node), K3s *agent process* (processo di servizio speciale utile alle operazioni di join/config iniziali, in esecuzione su entrambe le tipologie di Node, *Server* e *Agent*). Rispetto a KubeEdge, il vantaggio evidente di tale soluzione, dimostrato sia dal suo deployment sia, in un secondo momento, dal deployment in essa di un'applicazione dimostrativa (di cui si parlerà nel seguito), risiede nell'essenzialità: infatti, le funzionalità e il funzionamento di tutti i componenti del control-plane K3s (*API Server*, *scheduler*, ecc.) sono incapsulati, rispettivamente, in un unico file binario e un unico processo che può flessibilmente essere installato come servizio sulla macchina (virtuale o fisica che sia) GNU/Linux ospitante.

Attraverso il deployment dell'applicazione dimostrativa prima accennata, condotta sul test-bed dimostrativo costituito da un cluster K3s mono-master molto essenziale installato su una macchina server bare-metal, si è voluto dimostrare che un Pod possa servire non solo a offrire un servizio esposto dal cluster K3s (come avviene nei tipici casi di web portal, web application e affini, i cui processi server, organizzati in microservizi containerizzati, incapsulati in Pod e orchestrati, inviano i contenuti applicativi ai client che vi si connettono) ma, piuttosto, a *prelevare dati* dall'esterno del cluster di orchestrazione. Ciò è concettualmente simile al caso dell'applicazione (containerizzata) del componente funzionale PMU che, nel caso reale, acquisisce dai nodi della power grid i valori di misura relativi alle metriche del dominio elettro-energetico attraverso la sensoristica connessa al nodo di computing di orchestrazione ed è concepibile come l'esecuzione della funzione software virtualizzata (*light-virt*) e orchestrata su un dispositivo IED costituito da HW general-purpose a basso costo (es., Raspberry-Pi, Arduino, ecc.) che integra sensoristica on-board connessa alle linee di potenza della power grid e che le offre il supporto di computing necessario a generare il *synchrophasor data-stream*.

Per cui, a titolo esemplificativo, si è fatto ricorso a un'applicazione prototipale di acquisizione di stream video[7], organizzata secondo i seguenti tre micro-servizi componenti:

1. client di acquisizione del flusso video;
2. front-end;
3. back-end di elaborazione del flusso video.

Il componente client provvede all'acquisizione dei flussi video provenienti dalle webcam fisicamente connesse al nodo del cluster di orchestrazione, li invia al componente di front-end che, quindi, raccoglie tutti i flussi generati delle sorgenti video e, a sua volta, li trasmette frame-by-frame al back-end. Quest'ultimo consiste in una rete neurale di object recognition, che elabora i frame ricevuti dal front-end restituendo metadati che creano dei boxes intorno agli oggetti riconosciuti nell'immagine del frame. Il componente front-end mette quindi insieme i frame e li espone sequenzialmente su una pagina web, multiplando opportunamente le sorgenti video. Al di là dei dettagli funzionali e implementativi interni dell'applicazione (comunque marginali ai fini di questa trattazione), l'esperimento condotto ha dimostrato la possibilità di:

- ✓ agganciare un device fisico all'interno di un container applicativo - cioè, nello specifico, rendere visibile e utilizzabile dal container applicativo un dispositivo ad esso esterno, connesso al sistema host sul quale esegue il container applicativo del modulo client;
- ✓ orchestrare in K3s, incapsulandolo entro un Pod, il container applicativo al cui interno è stato agganciato un device fisico - cioè, nello specifico, effettuare il deployment nel cluster K3s (in forma di Pod) del container del modulo client avente accesso al device fisicamente connesso a un nodo di computing / worker node appartenente al cluster di orchestrazione.

Pertanto, l'obiettivo iniziale è consistito nell'eseguire il container applicativo del modulo client di acquisizione stream video lanciandolo sul nodo di computing (sistema host) in modalità "stand-alone" non orchestrata (cioè non associato agli altri micro-servizi componenti l'applicazione e non deployato sul cluster K3s), avendo previamente mappato il device fisico di acquisizione video streaming (webcam) nel container stesso. Quanto al primo step iniziale, è stato condotto il *build* della container-image del client (partendo dal Dockerfile disponibile, leggermente customizzato) necessaria a istanziarne ed eseguirne il relativo container, mappando poi in esso il device fisico (i.e. la webcam) in fase di esecuzione avendo sfruttato un'opzione apposita del CRE *Docker*. In seconda battuta, l'obiettivo finale di orchestrare in K3s il Pod incapsulante il container applicativo, quindi eseguire l'applicazione demo nella sua interezza, è stato raggiunto creando un apposito deployment (attraverso manifest file YAML) che, analogamente al caso del CRE *Docker* locale alla macchina host usata per raggiungere l'obiettivo iniziale, ha consentito di montare / mappare il

[7] Video capture demo application GitHub repository: https://github.com/mlavacca/cv_demo_webcam

device (fisicamente connesso a un nodo di computing / worker node del cluster K3s) nel Pod che incapsula il container applicativo.

7.3 Integrazione containerizzazione e orchestrazione

In seguito all'esperimento appena descritto, si è quindi proceduto alla fase di *integrazione* della *containerizzazione* delle applicazioni *PMU/iPDC* di simulazione dei componenti funzionali PMU/PDC discussa in sotto-sezione 7.1 con l'*orchestrazione* fin qui trattata.

7.3.1 Infrastruttura di orchestrazione

L'integrazione appena detta ha richiesto primariamente la definizione e l'implementazione dell'infrastruttura indispensabile al dispiegamento della soluzione di orchestrazione K3s, cioè dell'insieme degli elementi hardware e software necessari ad abilitare l'esecuzione dell'orchestratore con l'obiettivo di produrre un ambiente (test-bed) che emulasse il più fedelmente possibile l'infrastruttura FEC proposta nella Sezione 6. Tale infrastruttura ha incluso:

a. capacità di elaborazione / computing,
b. capacità di interconnessione di rete / networking,
c. capacità di archiviazione / storage,

naturalmente oltre a un'opportuna interfaccia che consentisse di accedere a tali risorse, principalmente virtualizzate. Le risorse virtuali realizzate hanno ragionevolmente rispecchiato una corrispondente infrastruttura fisica usualmente composta da server fisici (quali, ad es., NebbioloTech Fog node e/o macchine "white label" equipaggiate con opportune dotazioni di CPU e RAM), switch di rete e cluster di dispositivi di archiviazione / memoria di massa.

7.3.1.1 Infrastruttura di computing

La strategia implementativa dell'infrastruttura di computing necessaria a ospitare l'orchestratore K3s è consistita nel creare un solo cluster di orchestrazione, nel quale i nodi di computing sono rappresentati da VM (*full-virt*) ciascuna ospitante una componente di orchestrazione ("master" o "worker") a sé dedicata. Più in dettaglio, l'entità *control-plane* del cluster è costituita da un'unica VM sulla quale esegue il K3s *Server* Node: funge esclusivamente da master node, infatti su essa non esegue alcuna applicazione containerizzata di simulazione PMU/iPDC ma offre supporto di computing solo ai moduli tipici del control-plane Kubernetes (API Server, scheduler, DBMS dello stato del cluster, controller manager) descritti nella sotto-sezione 3.2.2. I containers delle applicazioni di simulazione (PMU/iPDC) relativi a ogni singolo componente funzionale (PMU/PDC) sono stati organizzati e deployati incapsulati in un singolo Pod come *sidecar containers*: ciascuno di tali Pod, che sia concettualmente sia concretamente rappresenta l'istanza di un componente funzionale (PMU/PDC) nella sua interezza e perciò detto *Pod CFC* per brevità, è stato quindi vincolato (per le ragioni esposte nel seguito, in sotto-sezione 7.4) ad eseguire sulla VM ad esso correlata ospitante il K3s *Agent* Node, la cui combinazione costituisce il worker node di processing/computing delle applicazioni containerizzate.

7.3.1.2 Infrastruttura di networking

Tutte le VMs del cluster K3s (create con la distribuzione Linux Ubuntu Server 20.04.1 LTS) sono state interconnesse mediante una infrastruttura di rete L2 virtualizzata costituita da un bridge (L2 virtual switching) che ha offerto la connettività backbone necessaria sia al traffico del cluster scambiato tra control-plane (VM con K3s *Server* Node / master node) e data-plane (VMs con K3s *Agent* Node / worker nodes), sia al traffico di management. L'indirizzamento L3 delle VMs (sottorete IPv4 che insiste sull'infrastruttura L2 suddetta) è stato gestito da un servizio DHCP; inoltre, la LIS di appartenenza delle VMs è risultata interconnessa ad internet tramite NAT in modo da garantire a esse l'accesso a repositories/registries (Ubuntu archives, DockerHub, GitHub, ecc.) necessario all'installazione di pacchetti/applicazioni accessorie utili al guest OS di ciascuna VM/host virtuale, dell'orchestratore K3s e al download delle indispensabili container-images relative alle applicazioni containerizzate (sia quelle di simulazione PMU/iPDC, sia quelle che creano l'ambiente grafico per la configurazione di queste ultime brevemente descritte nel seguito), previamente pushate su Docker registry pubblico. Quanto alla scelta del plug-in CNI che realizzasse nel cluster K3s le *Pod network* in modo che i container delle applicazioni di simulazione PMU/iPDC incapsulati in Pod fossero abilitati a scambiare traffico di rete tra loro e con l'esterno, nell'ordine sono stati vagliati: *Canal*, *Calico* e *Flannel*. La scelta è ricaduta su *Flannel* in virtù della semplicità implementativa interna di esso, che ha creato una rete overlay intra-cluster basata sul protocollo VXLAN.

7.3.1.3 Infrastruttura di storage

Lo storage dedicato alla soluzione di orchestrazione è stato concepito in maniera "ibrida" come composto da archiviazione su file system locale a ciascuna VM ospitante il proprio K3s *Agent* Node (per offrire persistenza ai dati applicativi del dominio elettro-energetico, ossia i sincrofasori componenti i synchrophasor data-streams) e da una VM dedicata all'esecuzione di un server NFS (containerizzato e orchestrato) che ha consentito di condividere, tra tutte le applicazioni (containerizzate e orchestrate) di simulazione PMU/iPDC, un file system di rete che rendesse flessibili le operazioni di salvataggio e caricamento dei file di ripristino delle configurazioni applicative necessarie al loro esercizio. L'implementazione dello storage è quindi consistita nel creare directory locali a ogni VM / K3s *Agent* Node, che hanno rappresentato l'interfaccia file system verso i supporti fisicamente utili all'archiviazione particolarmente necessaria ai container dei DBMS *MySQL* di ogni Pod CFC del componente funzionale PDC (Pod CFC PDC). Sotto il profilo logico, tale archiviazione è stata in seguito gestita mediante specifici oggetti Kubernetes (e relativi pattern/costrutti implementativi) aventi più alto livello d'astrazione e offerti dalla soluzione di orchestrazione K3s/Rancher secondo il paradigma dettagliato nella sotto-sezione 7.3.4 seguente.

È dunque utile introdurre, in Figura 9, la rappresentazione grafica schematica dell'infrastruttura di orchestrazione sopra descritta, che ne condensa i punti di vista architetturale e funzionale.

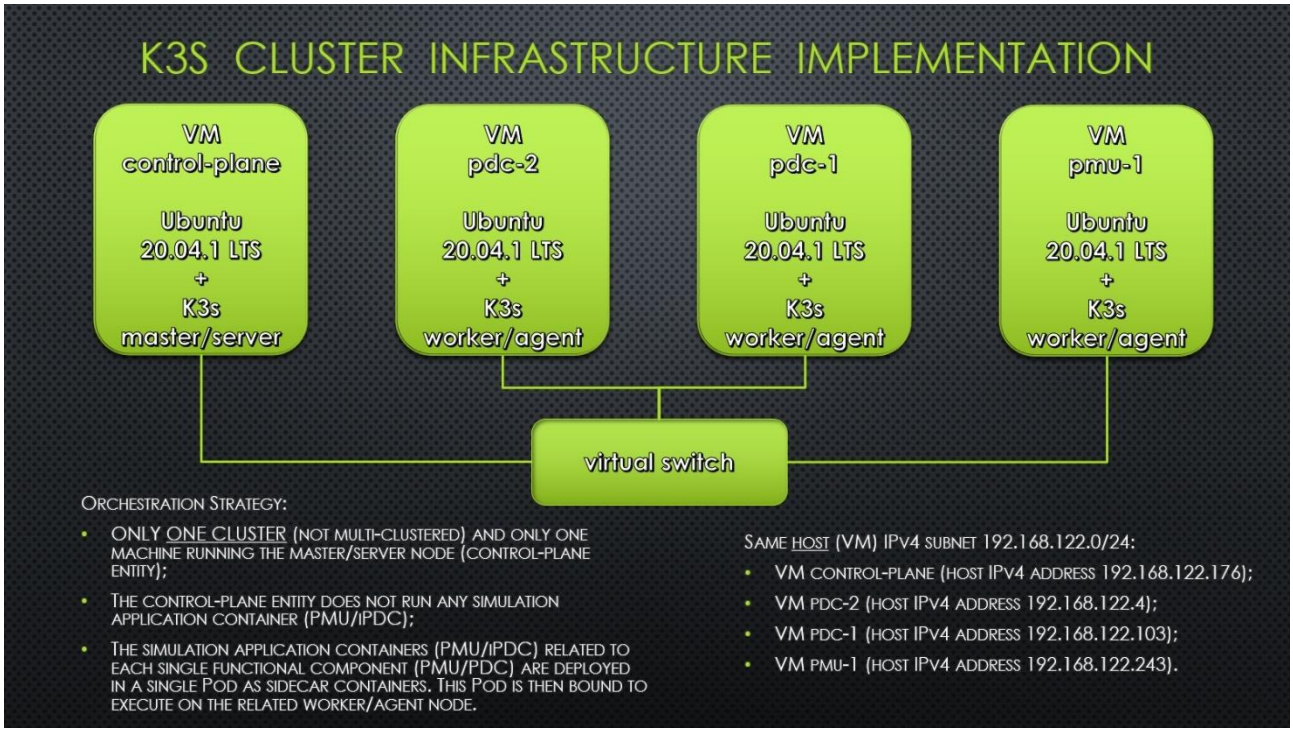

FIGURA 9 – SCHEMA IMPLEMENTATIVO DELL'INFRASTRUTTURA DI ORCHESTRAZIONE

7.3.2 Orchestrazione applicazioni PMU/iPDC: computing

Le applicazioni PMU e iPDC che simulano, rispettivamente, i componenti funzionali PMU e PDC, sono state sviluppate [Khandeparkar2012] per offrire all'operatore una GUI che ne semplificasse le operazioni di configurazione ed esercizio. Per tal motivo, si è reso essenziale ricorrere a un set di containers applicativi aggiuntivi che:

a. creasse l'ambiente desktop grafico necessario a renderizzare le finestre della GUI delle applicazioni containerizzate PMU/iPDC;
b. servisse all'operatore connesso al cluster l'ambiente desktop remoto indispensabile a interagire con le applicazioni stesse.

Pertanto, si è fatto ricorso a una sezione del progetto *CrownLabs* [CrownLabs2020] che ha fornito le funzionalità richieste nei punti a. e b. soprastanti mediante la triade di containers applicativi denominati *noVNC*, *WebSockify* e *TigerVNC* che, nel complesso, costituiscono il front-end grafico dell'applicazione containerizzata di simulazione PMU/iPDC. Più specificamente, ciascuno di tali container espleta la seguente funzione:

- *noVNC* istanzia l'immagine di un web server NGINX (protocol proxy) usato da un lato (servizio/back-end) per trasformare il protocollo tipico del VNC graphic server (flusso generato dal container *TigerVNC*) in un protocollo web-based (HTTP), dall'altro lato (utente/front-end) per servire all'operatore tale protocollo web-based che gli consenta di visualizzare il display remoto mediante web-browser;

- *WebSockify* da un lato incapsula il protocollo tipico del VNC graphic server (container *TigerVNC*) all'interno di un WebSocket, inoltrandolo (tramite il container *noVNC*) al web-browser dell'operatore e, dall'altro lato, trasforma/de-capsula il contenuto del WebSocket (i.e. il protocollo VNC graphic server) inviandolo al VNC graphic server attraverso un socket dati locale condiviso;
- *TigerVNC* esegue un desktop grafico virtuale implementato usando il VNC graphic server *tigerVNC*; include Fluxbox, cioè un window manager che offre un desktop grafico abilitando il rendering delle finestre.

La struttura del Pod CFC di ciascuna tipologia di componente funzionale (PMU/PDC) istanziato sul proprio K3s worker node è visibile in Figura 10, nella quale i containers delle applicazioni di simulazione PMU/iPDC del dominio elettro-energetico di interesse sono rappresentati in arancio, i containers aggiuntivi atti a offrire la GUI all'operatore in azzurro.

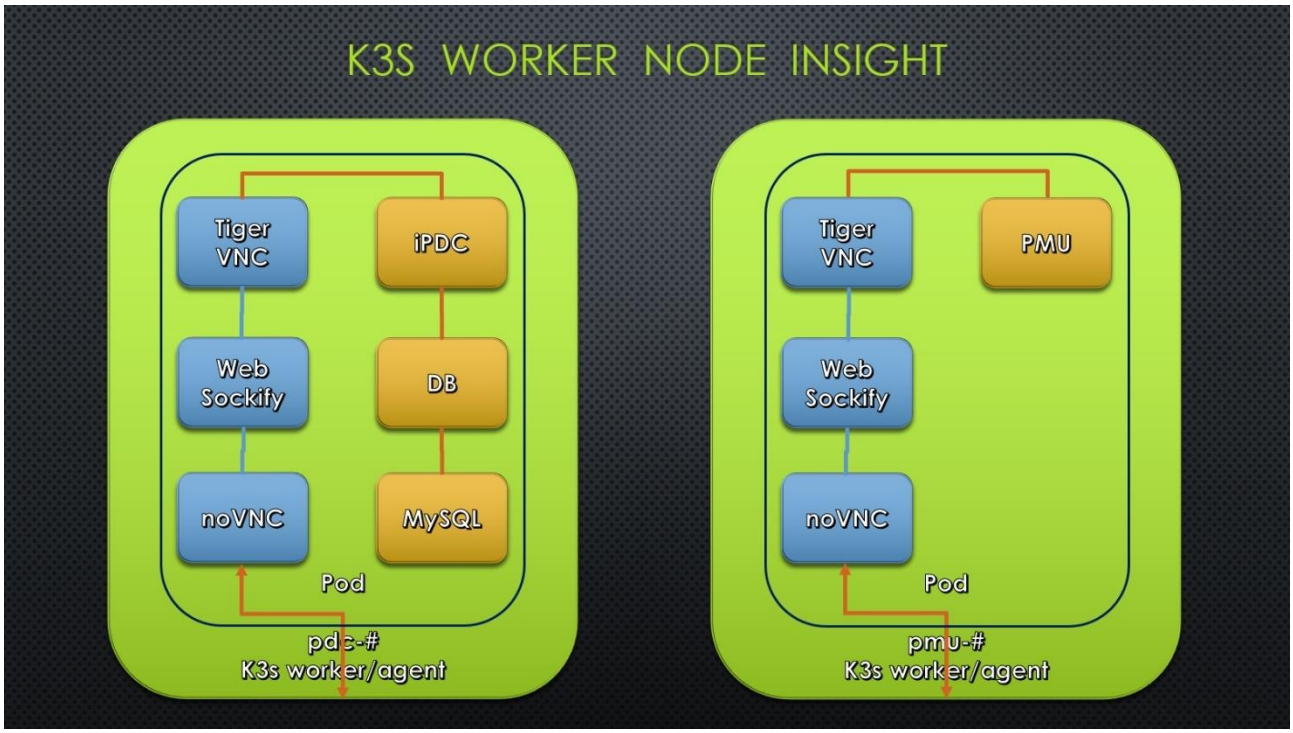

FIGURA 10 – STRUTTURA DEI POD CFC PMU/PDC CHE INCAPSULANO LE APPLICAZIONI DI SIMULAZIONE

Di fatto, non solo sotto il profilo concettuale e architetturale ma anche dal punto di vista implementativo e del computing, il CFC è rappresentato dall'aggregazione di tutti i containers applicativi in un'istanza dell'entità logica Pod, brevemente indicata con *Pod CFC* come già accennato.

Dunque, considerando l'aspetto del *computing* della soluzione di orchestrazione, il dispiegamento di tutti i Pod CFC sul cluster K3s è stato compiuto ricorrendo alla tipologia di oggetto Kubernetes detto *Deployment*: ogni istanza Pod CFC è stata modellata mediante una corrispondente istanza dedicata/univoca di tale oggetto che provvede a crearla e istanziarla sulla VM del corrispondente

K3s worker node, e la cui struttura è specificata mediante l'usuale sintassi dei file manifest YAML utilizzati per implementare in modo strutturato e ordinato qualsiasi risorsa sul cluster di orchestrazione.

Per *Deployment* Kubernetes si intende un oggetto risorsa orchestrata che fornisce aggiornamenti dichiarativi alle applicazioni dispiegate nel cluster e consente di descriverne il ciclo di vita mediante la specifica di aspetti quali le container-images da utilizzare (*pull* dal repository DockerHub prima accennato), il numero di Pod necessari (detto *repliche*) e le modalità di aggiornamento. Come gli altri, un tale oggetto Kubernetes serve a indicare al sistema orchestrato in che modo le applicazioni containerizzate PMU/iPDC debbano essere eseguite nel cluster. Una volta creato l'oggetto deployment specifico del CFC PMU/PDC, il control-plane agisce per garantirne l'esistenza e per mantenerne lo stato desiderato nel cluster. Si è fatto ricorso al Deployment dei Pod CFC per evitare i potenziali rischi operativi derivanti dal dispiegamento e dall'aggiornamento manuale delle applicazioni containerizzate in genere (e, nello specifico, quelle del dominio di interesse) nel cluster. Infatti, ad es., per dispiegare inizialmente un'applicazione complessa è necessario ricorrere a comandi articolati e potenzialmente numerosi. Per aggiornare manualmente un'applicazione a nuova versione è necessario avviare la nuova versione del relativo Pod, arrestare quella precedente, attendere e verificare che la versione più recente sia stata avviata correttamente; in caso di errori, spesso è necessario effettuare il *rollback* alla versione obsoleta. L'esecuzione manuale di tutti questi passaggi è naturalmente prona a errori umani, rappresentando un rallentamento (o, addirittura, un ostacolo) operativo per l'operatore che, ad es., debba compiere un processo di update urgente a una nuova release software del CFC PMU/PDC. Ecco che l'oggetto Deployment tende ad automatizzare e rendere operativamente *ripetibili* i procedimenti appena esemplificati. Un Deployment garantisce quindi in ogni momento l'esecuzione e la disponibilità del numero previsto di Pod CFC.

7.3.3 Orchestrazione applicazioni PMU/iPDC: networking

Dal punto di vista della *raggiungibilità*, cioè considerando l'aspetto del *networking* dei CFC PMU/PDC e alla luce di quanto rappresentato in Figura 10, il traffico applicativo che si origina da e termina in ciascun Pod CFC è costituito perciò da due flussi distinti di input/output al Pod stesso: *i. flusso di servizio-management*, iniziato dall'operatore che accede da remoto alla GUI dell'applicazione PMU/iPDC voluta per configurarne i parametri operativi, e *ii. flusso di processo elettro-energetico*, iniziato dalle applicazioni PMU/iPDC quando opportunamente configurate per instaurare tra loro i canali di comunicazione necessari alla trasmissione dei synchrophasor data-stream (flussi di SDF) secondo la gerarchia già vista. Per gestire la raggiungibilità dei Pod CFC, ossia i flussi di traffico suddetti, si è fatto quindi ricorso all'implementazione Kubernetes dei c.d. *Services*, cioè una modalità astratta utile a esporre un'applicazione in esecuzione su un Pod (o su un set di essi) come servizio di rete. In Kubernetes (K8s/K3s) i Pod sono per definizione risorse non permanenti: ciascun Pod possiede il proprio indirizzo IP interno "effimero", esistente finché il Pod esiste (i.e. limitatamente al termine del ciclo di vita del Pod); per cui, mediante la specifica di un

Service, si riesce ad associare a esso (o al set di essi) un nome DNS univoco corrispondente a un "virtual IP" che non cambia finché il Service esiste (e, naturalmente, è noto al servizio DNS del cluster detto *CoreDNS*). In definitiva, un Service è un oggetto del cluster responsabile di esporre un'interfaccia logica al Pod (o set di Pod) che consenta l'accesso a quest'ultimo dall'interno della rete del cluster e/o da processi esterni al cluster.

Secondo tale semantica, si sono dunque creati due *servizi* per ciascun Pod CFC: *i. servizio di front-end*, necessario a garantire il flusso di servizio-management e *ii. servizio di back-end*, necessario a garantire il flusso di processo elettro-energetico. Nello specifico, in entrambi i casi *i.* e *ii.* a livello implementativo si è usato il tipo di Service detto *NodePort*, che espone le applicazioni containerizzate PMU/iPDC del Pod CFC sull'indirizzo IPv4 di ogni VM K3s worker node in corrispondenza di una porta statica (la NodePort appunto, il cui valore è impostabile nel range default 30000÷32767). È perciò possibile raggiungere i servizi NodePort front-end e back-end (i.e. la parte di servizio-management e la parte di processo dell'applicazione containerizzata di simulazione PMU/iPDC, rispettivamente) dall'esterno del cluster contattando la destinazione `<K3sWorkerNodeIPv4>:<ServiceNodePort>`.

7.3.4 Orchestrazione applicazioni PMU/iPDC: storage

In seguito all'implementazione del *computing* e del *networking* della soluzione di orchestrazione, si è posto il problema di garantire la persistenza dei dati applicativi di processo relativi alle metriche di misura del dominio elettro-energetico (cioè i sincrofasori estratti dai synchrophasor data-streams / stream SDF) ricevuti e aggregati dai Pod CFC PDC che incapsulano i container delle applicazioni containerizzate di simulazione *iPDC* e, soprattutto, del DBMS *MySQL* che rappresenta l'applicazione di interfaccia con lo storage. L'insieme di questi due microservizi, *iPDC* in combinazione con *MySQL* (si ricorda, eseguiti come sidecar containers entro lo stesso Pod CFC PDC), che impongono il requisito della persistenza (i.e. memorizzazione e archiviazione strutturata) dei dati applicativi, determina una applicazione containerizzata *con stato*, c.d. *stateful*. Il concetto speculare a quest'ultimo è quello di applicazione *senza stato*, c.d. *stateless*. È possibile considerare lo *stato* di un'applicazione come la condizione di esistenza, in un determinato istante, di un elemento di essa: ad es., dei suoi dati, esattamente come nel caso dei sincrofasori ricevuti da *iPDC*. La condizione *stateful* o *stateless* di un'applicazione dipende dalla durata del mantenimento dell'interazione con l'elemento stesso e dalle modalità di memorizzazione di questa informazione. Pertanto, nel caso di specie del Pod CFC PDC, è esattamente la necessità di *memorizzazione permanente* dei sincrofasori ricevuti a determinarne la condizione *stateful*. Tale requisito di archiviazione dei dati determina dunque l'esigenza di avere accesso a uno storage che li conservi in modo permanente.

Bisogna premettere che i file presenti sul file system UFS (ved. sotto-sezione 3.1.1.1) di qualsiasi container sono *effimeri*, il che presenta una forte criticità per applicazioni containerizzate organizzate a miscroservizi di tipo *stateful* come la combinazione di *iPDC* e *MySQL*. Il problema principale è costituito dalla perdita dei file suddetti (es., database di sincrofasori) quando un container vada in crash o, in generale, si arresti in modo anomalo. Nello scenario orchestrato

K8s/K3s, in caso di arresto anomalo di un container in un Pod, la *kubelet* del worker node del cluster riavvia sì una nuova istanza dello stesso container ma che, però, è caratterizzata dal "clean state", cioè dallo stato pulito del file system UFS esattamente come derivante dall'istanziazione della container-image. In pratica, qualsiasi file scritto sul file system locale al container durante l'esecuzione di esso non sopravvive al ciclo di vita (es., crash) del container stesso, a meno di adottare le soluzioni di storage esposte nel seguito. Un problema secondario si verifica quando è necessario condividere file/directory tra sidecar containers appartenenti allo stesso Pod. La soluzione a entrambi i potenziali problemi visti, comunque riscontrati nel caso di specie del Pod CFC PDC, è proposta dall'implementazione Kubernetes dell'astrazione chiamata *Volume*.

In sostanza, un Volume è assimilabile a una directory, magari popolata da dati al suo interno, resa accessibile ai container incapsulati in un Pod. Il modo in cui tale "directory" venga creata, il supporto fisico sul quale insista e i contenuti di essa sono determinati dal particolare tipo di Volume scelto. Kubernetes supporta molti tipi di *Volume* e un Pod può utilizzarne contemporaneamente un numero arbitrario. Le varie tipologie di Volume sono comunque categorizzabili in due macro-famiglie fondamentali:

1. *effimeri*, aventi esistenza limitata alla durata del ciclo di vita dei Pod che li usino, garantendo persistenza transitoria dei dati dei Pod;
2. *persistenti*, aventi esistenza non limitata alla durata del ciclo di vita dei Pod che li usino, garantendo persistenza permanente dei dati dei Pod.

Di conseguenza, un Volume sopravvive comunque al ciclo di vita di qualsiasi container eseguito all'interno di un Pod, per cui i dati vengono conservati alla creazione di nuove istanze "pulite" del container stesso (ad es., in seguito a crash e restart). In altre parole, qualsiasi tipologia di Volume sopravvive sempre alla fine dell'esistenza di un container incapsulato in un Pod. Invece, quando un Pod cessa di esistere giungendo al termine del proprio ciclo di vita, Kubernetes *distrugge i volumi effimeri* e *non distrugge i volumi persistenti*.

Va detto che, nell'ambito dell'orchestrazione, la gestione dello *storage* è un aspetto distinto dalla gestione delle istanze di computing, indipendentemente dalla granularità con cui queste ultime vengano viste (che vada dalla singola applicazione containerizzata, al Pod, fino al worker node). In proposito, il sottosistema *PersistentVolume* di Kubernetes fornisce agli amministratori un'API che astrae e disaccoppia i dettagli su come lo storage venga fornito alle applicazioni containerizzate da come venga consumato da esse. Allo scopo di produrre la persistenza voluta, si è fatto perciò ricorso a due fondamentali risorse API di Kubernetes: gli oggetti *PersistentVolume* (*PV*) e *PersistentVolumeClaim* (*PVC*). Un PV è una parte di archiviazione nel cluster il cui provisioning può essere amministrato staticamente e manualmente dagli amministratori oppure dinamicamente utilizzando le risorse *StorageClasses* (descrizioni di "classi" di archiviazione mappate, ad es., su livelli di QoS, policy di back-up/DR, ecc.). In pratica, un PV è una risorsa nel cluster esattamente come lo è un worker node e può essere visto come un elemento plug-in, cioè come un volume di

archiviazione collegabile a richiesta dai Pod. La caratteristica distintiva del PV, che garantisce la persistenza ai Pod CFC PDC (che contengono le applicazioni containerizzate *iPDC* e *MySQL* stateful), è di possedere un ciclo di vita totalmente indipendente da quello di ogni singolo Pod CFC che lo utilizzi: se il container applicativo (*iPDC* e/o *MySQL*) cessa la propria esecuzione e/o il relativo Pod CFC PDC termina il proprio ciclo di vita (ad es., a causa di crash), i dati memorizzati sul backing-store attraverso il PV permangono. L'oggetto PV acquisisce i dettagli implementativi dello storage (ad es., iSCSI, NFS, CephFS, sistemi di storage proprietari dei Cloud provider, ecc.) e offre un'interfaccia astratta di questi ai Pod che lo necessitano: si tratta cioè di un layer logico che disaccoppia l'implementazione dello storage da come esso venga acceduto e consumato dai Pod di applicazioni stateful che richiedono persistenza, come il Pod CFC PDC. Un PVC (*PersistentVolumeClaim*) è, invece, un'effettiva richiesta di storage / archiviazione da parte di un Pod: un PVC offre al Pod l'effettiva possibilità di consumare le risorse di storage stabilite dal PV. Grazie ai PVCs, i Pod possono richiedere specifiche dimensioni e modalità di accesso a tali risorse di archiviazione, secondo un paradigma che rende l'implementazione dello storage sul cluster maggiormente flessibile. In definitiva, sotto il profilo logico può essere fatta la seguente considerazione generale: dal punto di vista del *computing*, i Pod consumano direttamente le risorse del worker node (CPU e RAM) richiedendole secondo livelli specifici; dal punto di vista dello *storage*, i PVC consumano direttamente le risorse PV (volumi persistenti) richiedendole secondo dimensioni e modalità di accesso specifiche, offrendo perciò ai Pod un'interfaccia astratta utile all'uso dei volumi di storage persistenti senza esporne i dettagli implementativi. Alla luce dell'analisi soprastante, per i Pod CFC PDC al momento si è implementata una soluzione di archiviazione articolata come segue:

a. alcuni Volumes effimeri necessari alla condivisione di path locali e socket di comunicazione tra i containers *TigerVNC*, *DB* e *MySQL*;
b. un PV *locale* alla VM ospitante il K3s worker node che, mediante corrispondente PVC invocato nei Deployment dei Pod CFC PDC, consente la persistenza dei sincrofasori ricevuti e archiviati nel database gestito dal container *MySQL*;
c. un PV *NFS* utile a produrre, invece, la persistenza distribuita delle configurazioni di tutte le applicazioni containerizzate di simulazione (PMU/iPDC) che, mediante corrispondente PVC invocato nei Deployment sia dei Pod CFC PMU sia dei Pod CFC PDC, ha abilitato la necessaria flessibilità nelle procedure di rispristino operativo delle funzionalità applicative di processo (esecuzione PMU/PDC) al verificarsi del crash dei container delle rispettive applicazioni PMU/iPDC.

7.4 Resilienza della soluzione di orchestrazione

Le applicazioni di simulazione *PMU* e *iPDC* rappresentano gli elementi operazionali dell'attuale definizione concettuale di CFC (PMU e PDC, rispettivamente) elaborata nell'ambito di questa tesi. La loro containerizzazione è stata condotta con l'obiettivo del deployment sperimentale in ambiente orchestrato Kubernetes (particolarmente K3s/Rancher). Tuttavia, essendo state codificate in tempi molto antecedenti il recente paradigma di sviluppo e deployment *cloud-native*, hanno presentato

limitazioni ineludibili che non hanno permesso di sfruttare pienamente le potenzialità di resilienza offerte dagli orchestratori. Il design originario delle applicazioni *PMU* e *iPDC* ha previsto che esse venissero eseguite come normali processi software quando installate in uno scenario simulativo *localhost*. Nello specifico, *iPDC* è stata progettata come applicazione composita, formata dal modulo *iPDC* omonimo e dal modulo *DB* che rappresenta l'interfaccia applicativa con il modulo *MySQL*: in particolare, la comunicazione originale tra *DB* e *MySQL* è stata realizzata mediante un path condiviso presente sulla macchina *localhost*. Si è reso quindi necessario condividere tale path anche in contesto orchestrato, mediante gli oggetti tipici di Kubernetes (Volumes). Da ciò si desume che, in fase di deployment nel cluster K3s, non è stato possibile disaccoppiare in alcun modo *DB* da *MySQL* - a meno di riprogettare in toto il modulo *DB*. Appare dunque evidente come la ragione di condensare tutti i containers applicativi in un'unica entità logica d'esecuzione (Pod CFC) organizzandoli come *sidecar* sia stata duplice: da un lato, di ordine pratico, per soddisfare il vincolo appena descritto (evitando di riscrivere il codice del modulo *DB*); d'altro canto, di ordine concettuale, per assumere una definizione integrata del CFC come costituito dal consolidamento di tutti i suoi componenti applicativi (i.e. tutti i sidecar containers).

Come detto, la coppia *DB* / *MySQL* implica il tipo *stateful* dell'applicazione containerizzata di simulazione *iPDC*, richiedendo il requisito fondamentale della *persistenza* a ciascuna istanza del Pod CFC PDC relativa. La combinazione di tale requisito con l'accoppiamento imprescindibile *DB* / *MySQL* ha dunque imposto il vincolo che ogni istanza del Pod CFC PDC sia stata eseguita sulla VM ad essa correlata ospitante il K3s *Agent* Node: di fatto, quindi, nel deployment si è dovuto schedulare staticamente ogni Pod CFC PDC a un proprio worker node di computing dedicato, archiviando i dati applicativi (sincrofasori) localmente al nodo, per cui il relativo database gestito da *MySQL* insiste permanentemente (attraverso l'impego degli oggetti Kubernetes PV e PVC) in un path locale alla VM che ospita il K3s worker node.

Allo stato attuale di sviluppo della soluzione, si è così raggiunto un *trade-off* che contempla sia l'esecuzione del Pod CFC PDC inteso come unica entità logica comprendente tutti i suoi componenti applicativi, sia la necessità del Pod di archiviare permanente i dati di processo elettro-energetico, naturalmente dovendo imporre un inevitabile limite alla resilienza applicativa complessiva dell'intero sistema. Tale limite si concretizza nel caso in cui il K3s worker node dedicato all'esecuzione di una particolare istanza del Pod CFC PDC dovesse non essere più raggiungibile (es., a causa di crash, network down, ecc., come ipotizzato nell'analisi degli Eventi #1 e #2 nelle sotto-sezioni 5.2 e 5.3). Infatti, al presente, essendo il re-scheduling del Pod CFC PDC vincolato al K3s worker node ad esso dedicato, è fondamentale disporre di ulteriori worker node ridondati e dedicati all'esecuzione di altrettante istanze *idle* (attive ma non ancora configurate) del Pod CFC PDC, che vengano opportunamente configurate dall'operatore (ricorrendo agli stessi file di configurazione archiviati e condivisi nella share NFS, che ne consentono il ripristino *seamless*) allorquando si verifichi l'evento critico suddetto.

Una possibile evoluzione della soluzione attuale, naturalmente a patto di praticare le dovute customizzazioni al software del modulo *DB* e nell'ottica di ottenere maggiore flessibilità e resilienza generale del sistema orchestrato (rilassando quindi i vincoli sia sotto l'aspetto del computing, sia dello storage prima visti), può essere costituita dall'impiego della piattaforma open-source di archiviazione distribuita *Ceph* [Ceph]. È un sistema di SDS (*Software Defined Storage*) basato su "oggetti", che garantisce supporto anche ai più tradizionali storage a blocchi e file system in un unico cluster di storage unificato, che da un lato lo rendono flessibile e altamente affidabile; dall'altro, molto complesso e ricco di componenti che richiedono installazione, configurazione e gestione accurate. I punti di forza fondamentali di Ceph consistono nelle sue caratteristiche di modularità, portabilità, scalabilità, resilienza, performance e self-healing (autorigenerazione). Ceph abilita la migrazione (spostamento) di elevatissime quantità di dati e oggetti di archiviazione (es., file) che richiedono persistenza quando gestiti dal relativo cluster, in modo da salvarne lo stato di memorizzazione desiderato. Il fatto che Ceph sia un sistema di archiviazione definito dal software offre grande flessibilità rispetto all'hardware, al SO e, soprattutto, alla posizione geografica in cui si collocano i nodi di computing/storage (avente granularità che va da un singolo rack, a un data center o anche a una regione DC). Infatti, esso implementa un'architettura core *object-based* che disaccoppia totalmente il SW di storage dal relativo HW: ciò consente di scalare la capacità di archiviazione in funzione delle esigenze di tipici scenari di produzione senza dover acquistare componenti HW / supporti backing-store proprietari ma economici ("white label"). Nel caso di specie dello scenario orchestrato Kubernetes, ad esempio, qualsiasi Pod può montare un Volume CephFS (in capo al sistema di storage Ceph) già esistente e magari pre-popolato: l'obiettivo potrebbe consistere nel condividere dati permanenti tra i Pod (scrittori multipli simultanei) sui quali il Volume CephFS è montato, in quanto il contenuto ne è preservato anche al termine del ciclo di vita dei Pod (in tal caso, il Volume viene semplicemente smontato).

Un sistema di storage SDS come Ceph può essere installato ed eseguito sia sul SO di una macchina host bare-metal, sia sul guest OS di una VM. Addirittura, i moduli Ceph possono essere containerizzati, permettendone così l'integrazione e la gestione su un'infrastruttura cluster orchestrata (bare-metal, virtuale o Cloud che sia).

Per fornire un'idea della complessità di tale soluzione, esiste un esempio[8] di deployment di un cluster di storage Ceph containerizzato in un cluster Kubernetes, che ne consente l'orchestrazione aggiungendo ai componenti Ceph tutte le feature offerte da un ambiente orchestrato.

Inoltre, un sistema di archiviazione distribuito come Ceph può essere abbinato a *Rook* [Rook], che ne estende le potenzialità trasformandolo in un *servizio di storage* autogestito, auto-scalabile e

[8] Containerized Ceph cluster on Kubernetes cluster: https://github.com/ceph/ceph-container/tree/master/examples/kubernetes

autorigenerante. Rook è un software open-source e cloud-native utile a orchestrare sistemi di storage, che rappresenta un framework di supporto alle soluzioni di archiviazione come Ceph.

Lo scopo di Rook è quello di automatizzare le attività di gestione e amministrazione del cluster di storage Ceph: deployment, avvio automatico, configurazione, provisioning, scaling, aggiornamento, migrazione, DR, monitoraggio e gestione delle risorse. Esso necessita della piattaforma Kubernetes per fornire questi servizi, facendo uso di un Operator Kubernetes dedicato al provider di archiviazione Ceph.

8 Prossimi sviluppi e conclusioni

Il lavoro condensato in questa tesi dimostra, quindi, come le soluzioni di tipo virtualizzato e orchestrato rappresentino di fatto un'alternativa all'attuale utilizzo di oggetti fisici dedicati, preinstallati con l'opportuno software.

A fronte di questo primo risultato, è stata definita l'architettura di massima del sistema, composta da nodi eterogenei, orchestrati, disposti su più livelli (dispositivi Edge, nodi Fog, Cloud) e si è proceduto a creare i componenti fondamentali necessari all'infrastruttura software (container Docker delle funzioni principali; analisi delle caratteristiche dei sistemi di orchestrazione KubeEdge e K3s/Rancher basati su Kubernetes, con installazioni sperimentali).

Il lavoro futuro si orienterà nella direzione delineata dai seguenti step evolutivi:

- definizione dettagliata dei componenti HW/SW e della loro dislocazione geografica. Questi componenti dovranno includere (1) le capacità SDN dell'infrastruttura di rete, (2) ipotesi più evolute di deployment dei principali componenti coinvolti (PMU/PDC, computing node) anche in base al loro posizionamento geografico, in modo da ottenere risultati più avanzati in termini di *ridondanza* (capacità di operare in presenza di guasti sfruttando eventuali funzioni ridondate presenti in rete) e di *resilienza* (capacità di controllare la micro-grid a fronte della mancanza di alcuni componenti, ad es. il collegamento verso il Cloud);
- definizione dettagliata dell'architettura di comunicazione reale tra i vari componenti in contesti di produzione, possibilmente impiegando tecnologie *event bus* (es., Apache Kafka). A titolo esemplificativo, la ridondanza di un Pod CFC PDC (es., assumendo due repliche, PDC1 e PDC2) è vana se un Pod CFC PMU invia il suo flusso dati ad un indirizzo IP predefinito sul quale risponde una sola istanza (es., PDC1), in quanto la ridondanza, pur prevista a livello architetturale, non può essere sfruttata;
- implementazione di una soluzione prototipale realizzante l'architettura HW/SW e di comunicazione definita al punto precedente;
- garantire maggiore resilienza dei servizi, attraverso l'implementazione di soluzioni di storage avanzate (es., Ceph/Rook) che consentano di disaccoppiare totalmente le applicazioni del dominio elettro-energetico (Pod CFC PDC) dai loro dati che necessitano di persistenza e, parimenti, rendendole il più possibile *stateless*, consentendone dunque il maggior grado possibile di flessibilità nel re-scheduling sui worker node del cluster di orchestrazione;
- analisi e misura delle prestazioni ottenibili e della risposta dell'architettura agli eventi critici definiti nella Sezione 5.

9 References

[LXC] "*LXC - Linux Containers*". linuxcontainers.org. Retrieved 2020-12-21. https://linuxcontainers.org/downloads/

[Barbier2014] Barbier, Julien (June 9, 2014). "*It's Here: Docker 1.0*". Docker. Docker, Inc. Retrieved September 30, 2019. https://blog.docker.com/2014/06/its-here-docker-1-0/

[Thurrott2007] Thurrott, Paul. "*Windows Server Virtualization Preview*". Archived on 2007-10-11. Retrieved 2007-09-25. https://web.archive.org/web/20071011034732/http://www.winsupersite.com/showcase/viridian_preview.asp

[DockerDocs2017a] "*Overview of Docker Compose*". Docker Documentation. Docker, Inc. Retrieved July 6, 2017. https://docs.docker.com/compose/

[DockerDocs2017b] "*Docker Swarm*". Docker Documentation. Docker, Inc. Retrieved July 6, 2017.

[Sefraoui2021] Sefraoui, O., Aissaoui, M., & Eleuldj, M. (2012). *OpenStack: toward an open-source solution for cloud computing*. International Journal of Computer Applications, 55(3), 38-42.

[ETSI2016] ETSI, O. (2016). *Open source mano*. OSM home page. https://osm.etsi.org/

[Sabharwal2013] Sabharwal, N., & Shankar, R. (2013). *Apache CloudStack cloud computing*. Packt Publishing.

[Pousty2014] Pousty, S., & Miller, K. (2014). *Getting Started with OpenShift: A Guide for Impatient Beginners*. O'Reilly Media, Inc.

[Burns2016] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). *Borg, omega, and kubernetes*. Queue, 14(1), 70-93.

[Rancher] Rancher Labs - The Rancher Website. https://rancher.com/

[RancherK3s] Rancher Labs - K3s Lightweight Kubernetes. https://k3s.io/

[KubeEdge] The KubeEdge project Website. https://kubeedge.io/

[PES2011] IEEE Power and Energy Society, C37.118.1-2011 - *IEEE Standard for Synchrophasor Measurements for Power Systems*, IEEE Power and Energy Society.

[PES2018] IEEE Power and Energy Society, 60255-118-1-2018 - *IEEE/IEC International Standard - Measuring relays and protection equipment* - Part 118-1: Synchrophasor for power systems - Measurements, IEEE Power and Energy Society.

[PES2013] IEEE Power and Energy Society, IEEE C37.244-2013 - *IEEE Guide for Phasor Data Concentrator Requirements for Power System Protection, Control, and Monitoring*, IEEE Power and Energy Society, 2013.

[MSE2020] Ministero dello Sviluppo Economico, *Piano Nazionale Integrato per l'Energia e il Clima (PNIEC)*, 21 Gennaio 2020. https://www.mise.gov.it/images/stories/documenti/PNIEC_finale_17012020.pdf.

[Kansal2012] P. Kansal, A. Bose, *Bandwidth and Latency Requirements for Smart Transmission Grid Application*, IEEE TRANSACTIONS ON SMART GRID, vol. 3, n. 3, pp. 1344-1352, Set 2012.

[Bernardes2019] E.B.C. Bernardes, D. M. L. Filho, B.G. Batista et alt, *Fog Computing Model to Orchestrate the Consumption and Production of Energy in Microgrids*, sensors, 2019.

[OpenPDC] OpenPDC installation. https://github.com/GridProtectionAlliance/openPDC.

[iPDC] iPDC project repository. http://iitbpdc.sourceforge.net/.

[DockerHub] The public Docker container-image repository and library. https://hub.docker.com/.

[ApacheKafka] Apache Kafka open-source stream-processing and message delivery software. https://kafka.apache.org/.

[Khandeparkar2012] Khandeparkar, K., & Pandit, N. (2012). "*Design and Implementation of IEEE C37.118 based Phasor Data Concentrator & PMU Simulator for Wide Area Measurement System*". Archived on 2012-09-11. https://versaweb.dl.sourceforge.net/project/iitbpdc/iPDC%20-%20Technical%20Report.pdf.

[CrownLabs2020] Cucinella, F., Francescato, L., Iorio, M., & Risso, F. (2020). Politecnico di Torino, NetGroup research group, CrownLabs project, "*From VM to Docker*" project repository on GitHub. https://github.com/netgroup-polito/CrownLabs/tree/master/provisioning/containers/gui-common.

[Ceph] Ceph open-source distributed storage software platform. https://ceph.io/.

[Rook] Rook open-source cloud-native management system for Kubernetes storage. https://rook.io/.

10 Glossario

Termine / Acronimo	Descrizione
API	Application Programming Interface
ARM	Advanced RISC Machine
CAPEX	CAPital EXpenditure (spese in conto capitale)
CD	Continuous Delivery
CFC	Componente Funzionale Critico
CI	Continuous Integration
CNI	Container Networking Interface
CRD	Custom Resource Definition
CRE	Container Runtime Engine
CRI	Container Runtime Interface
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
DB	Database
DBMS	Database Management System
DC	Data Center
DERTF	Distributed Energy Resources Test Facility
DevOps	Development Operations
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DR	Disaster Recovery
E2E	End-to-End
FEC	Fog/Edge/Cloud computing
full-virt	Full Virtualization
GPS	Global Positioning System
GPU	Graphical Processing Unit
HA	High Availability

HTTP	HyperText Transfer Protocol
HW	Hardware
ICT	Information and Communication Technology
IEC	International Electrotechnical Commission
IED	Intelligent Electronic Device
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISO	International Standard Organization
IT	Information Technology
K8s	Kubernetes
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
L3	ISO/OSI Layer 3 (Network Layer)
L7	ISO/OSI Layer 7 (Application Layer)
LAN	Local Area Network
light-virt	Lightweight Virtualization
LXC	Linux Container
LTS	Long Term Support
µG	Micro Grid
MAC	Media Access Control
MG	Micro Grid
MQTT	Message Queue Telemetry Transport
NAT	Network Address Translation
NIC	Network Interface Card
OS	Operating System

OSI	Open Systems Interconnection
OT	Operation Technology
PDC	Phasor Data Concentrator
PG	Power Grid
PMU	Phasor Measurement Unit
PV	Persistent Volume
PVC	Persistent Volume Claim
QoS	Quality of Service
RAM	Random Access Memory
RAN	Radio Access Network
REST	Representational State Transfer
RISC	Reduced Instruction Set Computer
ROCOF	Rate of Charge of Frequency
RSE	Ricerca sul Sistema Energetico
RTT	Round Trip time
SDF	Syncrophasor Data Frame
SDK	Software Development Kit
SDN	Software Defined Networking
SDS	Software Defined Storage
SG	Smart Grid
SO	Sistema Operativo
SW	Software
TCP/IP	Transmission Control Protocol/Internet Protocol
UFS	Union File System
VCPU	Virtual CPU
VLAN	Virtual LAN
VM	Virtual Machine
YAML	Yet Another Markup Language

11 Appendice A: Cenni di configurazione

11.1 Configurazione dei containers delle applicazioni PMU/PDC

Questa sotto-sezione elenca e descrive i comandi principali necessari alla creazione dei container relativi alle applicazioni PMU/PDC del dominio elettro-energetico.

In seguito alla stesura di un apposito Dockerfile che descrive i componenti inclusi nella creazione della container-image desiderata (basata sulla distribuzione Linux Ubuntu 14.04.6 LTS), avviene la costruzione vera e propria (*build*) di quest'ultima tramite il seguente comando:

```
-   root@master:~# docker build –t [image_name] [folder]
```

in cui `image_name` rappresenta il nome della container-image creata e `folder` è la cartella che contiene il Dockerfile unitamente ad eventuali altri file (sotto-directory, archivi *.zip/*.tar, file di configurazione, script, ecc.) necessari alla creazione e all'esecuzione dell'ambiente containerizzato voluto. Tramite il comando

```
-   root@master:~# docker images --digests
```

è possibile elencare tutte le immagini presenti sul nodo di computing, quindi verificare se il comando `build` abbia avuto esito positivo individuando nella lista di output la container-image contraddistinta da `image_name` e basata su Linux Ubuntu 14.04.6 LTS.

È perciò possibile lanciare un container vero e proprio, istanza della container-image creata in precedenza, ad esempio eseguendo al suo interno una shell (es., `/bin/bash`):

```
-   root@master:~# docker run -it --name [container_name] [image_name] /bin/bash
```

dove `[container_name]` (specificato mediante l'opzione/flag `--name`) è il nome da assegnare al container prossimo all'esecuzione e `[image_name]` è la container-image base creata in precedenza. In seguito, con il container running e la shell in esecuzione al suo interno, è possibile eseguire l'insieme dei comandi necessari alla installazione di *Mono*, delle apposite librerie e dello script bash `install-openPDC.sh` che, una volta terminato, avrebbe dovuto compilare i sorgenti e creare il file eseguibile `openPDC.exe`, consentendo dunque l'esecuzione di esso mediante il seguente comando:

```
-   root@master:~# mono [installation_folder]/openPDC.exe
```

Va precisato che, durante il procedimento di cui sopra, sono stati riscontrati errori durante l'esecuzione dello script bash che impediscono il processo di creazione e installazione dell'applicativo *openPDC*. Esaminando il file di log di tale processo, gli errori generati sono stati

ricondotti a dipendenze mancanti e/o obsolete. Nonostante sia stata aperta una segnalazione di problema tecnico ("*issue*") sul repository GitHub del progetto *openPDC*, allo stato attuale le cause di tali mancanze e obsolescenze riscontrate sono ancora ignote, non avendo ricevuto alcun riscontro in merito.

È stata perciò provata l'installazione di *openPDC* versione 2.8 (latest version) su un container istanziato dalla container-image basata sulla distribuzione Linux Ubuntu 20.04.1 LTS (Focal Fossa). Per ottenere la container-image (già costruita) di tale distribuzione, il comando

```
-   root@master:~# docker pull ubuntu
```

ha permesso di eseguire il download progressivo degli strati software necessari (librerie, eseguibili, ecc.) dal repository pubblico DockerHub. Lo script bash `buildmono.sh`, posto nella sotto-cartella `Source`, è stato allora lanciato a seguito dell'installazione di *Mono* latest version (versione 6.12.0.107). A differenza del caso precedente, l'installazione è stata completata e l'applicativo installato correttamente ma, al momento dell'esecuzione di esso tramite *Mono*, è stata sollevata un'eccezione del tipo "FileNotFound". Infatti, sulla base degli errori riportati sulla shell, la libreria `GSF.Core` è risultata mancante, impedendo all'applicativo `openPDC.exe` di eseguire correttamente senza il lancio di eccezioni.

11.2 Installazione dell'orchestratore Kubernetes / KubeEdge

11.2.1 Requisiti di installazione

Di seguito si schematizzano i prerequisiti di installazione necessari a un deployment efficace completo, per ambienti di produzione, della soluzione KubeEdge.

- Cloud layer:
  - Cluster K8s vanilla prodromico;
  - Container Runtime Engine (scelto *Docker*, possibili scelte - comunque mutuamente esclusive - sono *containerd* e *CRI-O*);
  - Container Networking Interface plug-in (CNI plug-in) / Pod network (inizialmente scelto *Cilium*, seguito dal testing di *Flannel* e di *Calico*).
- Edge layer:
  - Container Runtime Engine (scelto *Docker*, necessaria uniformità con il Cloud layer);
  - MQTT Broker (*Mosquitto*).
- Device layer:
  - IED / hardware IoT, es. Raspberry-Pi + sensoristica on-board.

Il test-bed reale implementato per il deployment della soluzione KubeEdge è costituito, per ragioni pratiche e di agilità, da un ambiente virtuale di full-virtualization (*KVM*, hypervisor type 1) che

supporta il provisioning di tre VM con SO guest Ubuntu server 20.04.1 LTS interconnesse a L3 (routing), i cui possibili requisiti funzionali, di configurazione e dimensionamento hardware virtuale minimo sono riportati di seguito.

VM KubeEdge Master Node

Questa VM ospita il control-plane di un cluster K8s.

Memoria: ≥ 2 GB di RAM;

Processing: ≥ 2 VCPU;

NIC:

- 1 NIC (verso default gateway): Connettività Internet per scaricare/estrarre immagini container e scaricare/installare pacchetti software necessari alle installazioni;
- 1 NIC (verso VM Router): Connettività di rete completa con la VM KubeEdge Worker Node (su sottorete privata o pubblica);

Hostname, **indirizzi MAC** e **product uuid** univoci all'interno del cluster.

Swap disabilitata affinché il componente *kubelet* possa eseguire correttamente.

VM Router

Distro usata per dimostrare il routing L3 tra i nodi KubeEdge.

Memoria: ≥512 MB di RAM;

Processing: ≥1 VCPU;

NIC:

1. 1 NIC (verso default gateway): Connettività Internet: per scaricare/installare eventuali pacchetti software accessori;
2. 2 NIC (verso VM KubeEdge Master Node e KubeEdge Worker Node): Connettività di rete completa con le VM/nodi KubeEdge del cluster (su sottorete privata o pubblica).

VM KubeEdge Worker Node

Simula un IED/hardware IOT general-purpose.

Memoria: ≥ 2 GB di RAM;

Processing: ≥ 2 VCPU;

NIC:

1. 1 NIC (verso default gateway): Connettività Internet per scaricare/estrarre immagini container e scaricare/installare pacchetti software necessari alle installazioni;
2. 1 NIC (verso VM Router): Connettività di rete completa con la VM KubeEdge Master Node (su sottorete privata o pubblica).

Il SO Ubuntu 20.04.1 LTS è stato scelto perché è tra le distribuzioni Linux ufficialmente supportate dai binary tool `kubeadm/keadm` usati, rispettivamente, per condurre l'installazione e la contestuale primissima configurazione dei cluster di orchestrazione Kubernetes/KubeEdge.

11.2.2 Realizzazione dell'infrastruttura Edge/Cloud con KubeEdge

Nella soluzione KubeEdge, le tre VM suddette vengono utilizzate secondo i ruoli riportati di seguito, sulla base dei quali sono stati installati i pacchetti necessari al deployment della soluzione.

VM KUBEEDGE MASTER NODE

È la VM maggiormente performante, sulla quale, nell'ordine, vengono installati i seguenti software necessari all'esecuzione di tutti i componenti Cloud layer e configurato il SO host:

1. Configurazione delle due NIC, la prima in DHCP verso il default gateway, la seconda staticamente su sottorete privata verso VM Router con route statica per raggiungere la VM KubeEdge Worker Node:

```
root@kemaster:~# nano /etc/netplan/00-installer-config.yaml
root@kemaster:~# netplan apply
```

2. Installazione dei pacchetti preliminari:

```
root@kemaster:~# apt -y install git curl wget kubeadm kubectl kubelet
root@kemaster:~# apt-mark hold kubeadm kubectl kubelet
```

3. Disabilitazione dello swap sul SO host per consentire a *kubelet* di eseguire:

```
root@kemaster:~# sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
root@kemaster:~# swapoff -a
```

4. Configurazione di `sysctl`:

```
root@kemaster:~# modprobe overlay
root@kemaster:~# sudo modprobe br_netfilter
root@kemaster:~# sudo tee /etc/sysctl.d/kubernetes.conf<<EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
```

```
root@kemaster:~# sysctl --system
```

5. Installazione e configurazione del CRE Docker:

```
root@kemaster:~# apt install -y containerd.io docker-ce docker-ce-cli
root@kemaster:~# mkdir -p /etc/systemd/system/docker.service.d
root@kemaster:~# tee /etc/docker/daemon.json <<EOF
{
 "exec-opts": ["native.cgroupdriver=systemd"],
 "log-driver": "json-file",
 "log-opts": {
  "max-size": "100m"
 },
 "storage-driver": "overlay2"
}
EOF
root@kemaster:~# systemctl daemon-reload
root@kemaster:~# systemctl restart docker
root@kemaster:~# systemctl enable docker
```

6. Inizializzazione del control-plane Kubernetes (K8s master node) con kubeadm init:

```
root@kemaster:~# systemctl enable kubelet
root@kemaster:~# kubeadm config images pull
root@kemaster:~# kubeadm init --pod-network-cidr=10.10.0.0/16 \
--control-plane-endpoint=k8s-cluster --service-cidr=10.20.0.0/16 --token-ttl 0
root@kemaster:~# export KUBECONFIG=/etc/kubernetes/admin.conf
root@kemaster:~# kubectl cluster-info
root@kemaster:~# watch kubectl get pods --all-namespaces –o wide
root@kemaster:~# kubectl get nodes -o wide
```

7. Installazione del plug-in CNI (Pod network) Cilium in modo che i Pod possano comunicare tra loro e il DNS del cluster (CoreDNS) possa partire:

```
root@kemaster:~# kubectl apply -f \
https://raw.githubusercontent.com/cilium/cilium/v1.9/install/kubernetes/quick-
install.yaml
root@kemaster:~# watch kubectl get pods --all-namespaces –o wide
```

Il Deployment del modulo KubeEdge *CloudCore* attraverso il binary tool *keadm* richiede i seguenti passaggi:

1. Scaricare e installare il tool *keadm*:

```
root@kemaster:~# wget https://github.com/kubeedge/kubeedge/releases/download/v1.5.0/keadm-
v1.5.0-linux-amd64.tar.gz
root@kemaster:~# tar -xvzf keadm-v1.5.0-linux-amd64.tar.gz
```

2. Inizializzare il modulo CloudCore con *keadm init*:

```
root@kemaster:~/keadm-v1.5.0-linux-amd64/keadm# ./keadm init \
--kubeedge-version=1.5.0 --kube-config=/root/.kube/config \
--advertise-address=192.168.50.2 --domainname=kemaster
Kubernetes version verification passed, KubeEdge installation will start...
...
KubeEdge CloudCore is running, for logs visit:
/var/log/kubeedge/cloudcore.log
root@kemaster:~# tail –f /var/log/kubeedge/cloudcore.log
```

3. Ottenere il token mediante cui il Master Node/*CloudCore* approverà in seguito il join del Worker Node/*EdgeCore* al cluster stabilendo un canale bidirezionale sicuro con esso. Esecuzione del comando *keadm gettoken*:

```
root@kemaster:~/keadm-v1.5.0-linux-amd64/keadm# ./keadm gettoken \
--kube-config=/root/.kube/config
9ab9e420f364d49ecd379cdecaba6e883279b547f3a0de1d126351e569bc5865.eyJhbGciOiJIU-
zI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MDYyMjk4MDN9.NSStqlZyYK7k8Ou--
TtXLevfqm6w717x3jcw30gEQsA
```

In via sperimentale, viene configurazione ed abilitazione anche la funzionalità *kubectl logs*, propedeutica al corretto deployment del *metrics-server* (componente necessario all'auto-scaling). Essendo nella soluzione KubeEdge presente una versione molto customizzata e piuttosto mal documentata rispetto a quella "canonica" di K8s vanilla, al momento non è in esecuzione.

VM ROUTER

È la VM che abilita il routing L3 tra la sottorete privata IPv4 della VM KubeEdge Master Node e la sottorete privata IPv4 della VM KubeEdge Worker Node:

1. Configurazione delle tre NIC, la prima in DHCP verso il default gateway, la seconda staticamente su sottorete privata verso VM KubeEdge Master Node con route statica per raggiungere la VM KubeEdge Worker Node, la terza staticamente su sottorete privata verso VM KubeEdge Worker Node con route statica per raggiungere la VM KubeEdge Master Node:

```
root@kerouter:~# nano /etc/netplan/00-installer-config.yaml
root@kerouter:~# netplan apply
```

2. Abilitazione del packet forwarding a L3 per il protocollo IPv4:

```
root@kerouter:~# nano /etc/sysctl.conf
root@kerouter:~# sysctl -p
```

VM KUBEEDGE WORKER NODE

Si tratta della VM sulla quale, nell'ordine, vengono installati i seguenti software necessari all'esecuzione di tutti i componenti Edge layer e configurato il SO host:

1. Configurazione delle due NIC, la prima in DHCP verso il default gateway, la seconda staticamente su sottorete privata verso VM Router con route statica per raggiungere la VM KubeEdge Master Node:

```
root@kerouter:~# nano /etc/sysctl.conf
root@kerouter:~# sysctl -p
```

2. Installazione e configurazione CRE *Docker* (avendo cura di impostare il default *cgroup* driver come "`native.cgroupdriver=cgroupfs`" e non "`native.cgroupdriver=systemd`", pena la mancata esecuzione del modulo *EdgeCore*):

```
root@keworker:~# apt install -y containerd.io docker-ce docker-ce-cli
root@keworker:~# mkdir -p /etc/systemd/system/docker.service.d
root@keworker:~# tee /etc/docker/daemon.json <<EOF
{
 "exec-opts": ["native.cgroupdriver=cgroupfs"],
 "log-driver": "json-file",
 "log-opts": {
  "max-size": "100m"
 },
 "storage-driver": "overlay2"
}
EOF
root@keworker:~# systemctl daemon-reload
root@keworker:~# systemctl restart docker
root@keworker:~# systemctl enable docker
```

3. Deployment del modulo KubeEdge *EdgeCore* attraverso il binary tool *keadm*:
   i. Download e installazione del tool *keadm*:

```
root@keworker:~# wget https://github.com/kubeedge/kubeedge/releases/download/v1.5.0/keadm-v1.5.0-linux-amd64.tar.gz
root@keworker:~# tar -xvzf keadm-v1.5.0-linux-amd64.tar.gz
```

   ii. Installazione del broker server MQTT (*Mosquitto*):

```
root@keworker:~# apt -y install mosquitto
```

   iii. Inizializzazione del modulo EdgeCore con *keadm join*:

```
root@keworker:~/keadm-v1.5.0-linux-amd64/keadm# ./keadm join \ --cloudcore-ipport=192.168.50.2:10000 \
--edgenode-name=keworker --kubeedge-version=1.5.0 \ --token=9ab9e420f364d49ecd379cdecaba6e883279b547f3a0de1d126351e569bc5865.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MDYyMjk4MDN9.NSStqlZyYK7k8Ou--TtXLevfqm6w717x3jcw30gEQsA
Host has mosquit+ already installed and running. Hence skipping the installation steps!!!
```

```
...
KubeEdge EdgeCore is running, for logs visit:
/var/log/kubeedge/edgecore.log
root@keworker:~# tail –f /var/log/kubeedge/edgecore.log
root@keworker:~# journalctl –fu edgecore.service
```

11.3 Installazione dell'orchestratore K3s/Rancher

11.3.1 Requisiti di installazione

Di seguito si schematizzano i prerequisiti di installazione necessari a un deployment efficace completo, per ambienti di produzione, della soluzione K3s/Rancher.

- Cloud / Fog layer:
  - K3s Server Node (control-plane su VM Ubuntu server 20.04.1) c.d. K3s Master Node:
    - Di default, per consentire un deployment completo, lo script di installazione invoca (ed eventualmente installa, se non già presenti) le seguenti dipendenze:
      - DBMS *SQLite3*;
      - CRE *containerd*;
      - CNI plug-in *Flannel*;
      - Cluster DNS service *CoreDNS*;
      - Host utility (quali *iptables*, *socat*, ecc.);
      - Ingress Controller *Traefik*;
      - Load Balancer service *Klipper*;
    - Inoltre, in generale K3s supporta e consente anche il deployment custom, comunque vincolato ai seguenti componenti in luogo di quelli default:
      - DBMS clusterizzati per large-scale deployment quali *MySQL*, *PostgreSQL*, *MariaDB*, *etcd* (installabili su cluster dedicati ed esterni alla VM K3s *Server* Node);
      - CRE *Docker*;
      - CNI plug-in *Canal* oppure *Calico*.
- Edge / Device layer:
  - K3s *Agent* Node (data-plane su VM Ubuntu server 20.04.1) c.d. K3s *Worker* Node:
    - Di default, per consentire un deployment completo, lo script di installazione invoca (ed eventualmente installa, se non già presenti) le seguenti dipendenze:
      - CRE *containerd* (eventualmente *Docker* in installazioni custom, comunque necessaria uniformità con il CRE installato in Cloud layer).

Si è perciò proceduto all'installazione "default" di tale soluzione orchestrata. È necessario precisare che, nel contesto implementativo, si definisce K3s *Server* Node una macchina (sia essa virtuale o bare-metal) sulla quale viene eseguito il comando `k3s server` (seguito dalle opportune opzioni/argomenti necessari a creare l'ambiente voluto). Le funzionalità e il funzionamento di tutti

i componenti del control-plane K3s (*API Server*, *scheduler*, ecc.) sono perciò incapsulati, rispettivamente, in un unico file binario e un unico processo.

Il test-bed reale implementato per il deployment della soluzione K3s/Rancher è costituito, per ragioni pratiche e di agilità, da un ambiente virtuale di full-virtualization (*KVM*, hypervisor type 1) che supporta il provisioning di due tipologie di VM con SO guest Ubuntu Server 20.04.1 LTS interconnesse a L2 (virtual switching offerto dalla piattaforma *KVM*), i cui possibili requisiti funzionali, di configurazione e dimensionamento hardware virtuale minimo sono riportati di seguito.

VM K3S MASTER NODE

Questa VM, unica della tipologia *Master*, ospita il control-plane del cluster K3s.

Memoria: ≥ 2 GB di RAM;

Processing: ≥ 2 VCPU;

NIC: 1 NIC (verso LIS privata IPv4 di appartenenza NAT-ed, default gateway), che consente:

- Connettività Internet per scaricare/estrarre le immagini dei container applicativi e scaricare/installare pacchetti software necessari alle installazioni;
- Connettività di rete completa (L2 virtual switch) con le VMs K3s Worker Node (su sottorete IPv4 / LIS privata).

Hostname, **indirizzi MAC** e **product uuid** univoci all'interno del cluster.

Swap disabilitata affinché il componente *kubelet* possa eseguire correttamente.

VMS K3S WORKER NODES

Queste VM, multiple della tipologia *Worker*, ospitano il data-plane del cluster K3s simulano gli IED/hardware IOT general-purpose (sui quali può eseguire l'applicazione containerizzata PMU) oppure i server / Fog node (sui quali può eseguire l'applicazione containerizzata PDC).

Memoria: ≥ 2 GB di RAM;

Processing: ≥ 2 VCPU;

NIC: 1 NIC (verso LIS privata IPv4 di appartenenza NAT-ed, default gateway), che consente:

- Connettività Internet per scaricare/estrarre immagini container e scaricare/installare pacchetti software necessari alle installazioni;
- Connettività di rete completa (L2 virtual switch) con la VM K3s Master Node (su sottorete IPv4 / LIS privata).

Il SO Ubuntu 20.04.1 LTS è stato scelto perché è tra le distribuzioni Linux ufficialmente supportate dal pacchetto binario necessario a condurre l'installazione e la contestuale primissima configurazione del cluster di orchestrazione K3s, che offre il deployment dei componenti K3s *Server* e K3s *Agent* con i rispettivi componenti interni già discussi nella sotto-sezione 7.2.2 "Implementazione K3s/Rancher".

11.3.2 Realizzazione dell'infrastruttura Edge/Cloud con K3s/Rancher

Nella soluzione K3s, le VM suddette vengono utilizzate secondo i ruoli riportati di seguito, sulla base dei quali sono stati installati i pacchetti necessari al deployment della soluzione.

VM K3S MASTER NODE

È la VM sulla quale, nell'ordine, vengono installati i seguenti software necessari all'esecuzione di tutti i componenti Cloud layer / Fog layer e configurato il SO host:

1. Verifica della configurazione automatica della NIC via DHCP lease (ottenuto da un DHCP server *dnsmasq*) sul link virtuale L2 verso il default gateway:

```
- root@k3smaster:~# ip link show
- root@k3smaster:~# ip address show
```

2. In seguito al setup preliminare dell'environment della VM (timezone, sincronizzazione del clock di sistema con i server NTP di INRIM, installazione di tool e pacchetti applicativi utili, ecc.), si procede a scaricare lo script necessario al deployment automatizzato del K3s *Server* Node:

```
- root@k3smaster:~# apt-get install -y curl git lsof vim wget
- root@k3smaster:~# curl -sfL https://get.k3s.io –o k3s_install.sh
```

3. Disabilitazione dello swap sul SO host per consentire a *kubelet* di eseguire:

```
- root@k3smaster:~# sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
- root@k3smaster:~# swapoff -a
```

4. Installazione (deployment automatizzato) del K3s *Server* Node con alcuni parametri default e altri customizzati mediante la combinazione di *environment variables* (variabili d'ambiente) note allo script di installazione con opzioni passate al comando `server`, che incorpora l'inizializzazione del control-plane Kubernetes (K3s master node):

```
- root@k3smaster:~# ./k3s_install.sh | INSTALL_K3S_NAME="master" sh -s - \
  server \
  -v 3 \
  --log $HOME/k3smaster.log \
  --alsologtostderr \
  --write-kubeconfig-mode 644 \
  --cluster-cidr 10.0.0.0/16
```

5. Copia del *node token*, generato dal Master Node, sul Worker Node, necessario a quest'ultimo per autenticarsi sul Master Node e compiere il join al cluster:

```
-   root@k3smaster:~# ls -lar /var/lib/rancher/k3s/server/node-token
-   root@k3smaster:~# scp /var/lib/rancher/k3s/server/node-token root@k3sworker:~/
```

6. Verifica della corretta esecuzione del processo *k3s server* sul Master Node:

```
-   root@k3smaster:~# ls -lar /etc/systemd/system/k3s-master.service
-   root@k3smaster:~# journalctl -fu k3s-master.service
```

7. Verifica della corretta installazione, configurazione ed esecuzione dell'orchestratore sul nodo:

```
-   root@k3smaster:~# tail -f k3smaster.log
-   root@k3smaster:~# k3s check-config
-   root@k3smaster:~# k3s kubectl get nodes -o wide
-   root@k3smaster:~# watch -c -n 0.5 k3s kubectl get pods --all-namespaces –o wide
```

VMs K3s Worker Nodes

Sono le VMs sulle quali, nell'ordine, vengono installati i seguenti software necessari all'esecuzione di tutti i componenti Edge layer / Device layer e configurato il SO host:

1. Verifica della configurazione automatica della NIC via DHCP lease (ottenuto da un DHCP server *dnsmasq*) ottenuto sul link virtuale L2 verso il default gateway:

```
-   root@k3sworker:~# ip link show
-   root@k3sworker:~# ip address show
```

2. In seguito al setup preliminare dell'environment della VM (timezone, sincronizzazione del clock di sistema con i server NTP di INRIM, installazione di tool e pacchetti applicativi utili, ecc.), si procede a scaricare lo script necessario al deployment automatizzato del K3s *Agent* Node:

```
-   root@k3sworker:~# apt-get install -y curl git lsof vim wget
-   root@k3sworker:~# curl -sfL https://get.k3s.io –o k3s_install.sh
```

3. Disabilitazione dello swap sul SO host per consentire a *kubelet* di eseguire:

```
-   root@k3sworker:~# sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
-   root@k3sworker:~# swapoff -a
```

4. Installazione (deployment automatizzato) del K3s *Agent* Node con alcuni parametri default e altri customizzati mediante la combinazione di *environment variables* (variabili d'ambiente)

note allo script di installazione con opzioni passate al comando `agent`, che incorpora l'inizializzazione del data-plane Kubernetes (K3s worker node):

```
-   root@k3sworker:~# ./k3s_install.sh | INSTALL_K3S_NAME="worker" sh -s - \
    agent \
    -v 3 \
    --log $HOME/k3sworker.log \
    --alsologtostderr \
    --server https://k3smaster:6443 \
    --token-file $HOME/node-token
```

5. Verifica della corretta esecuzione del processo *k3s agent* sul Worker Node:

```
-   root@k3sworker:~# ls -lar /etc/systemd/system/k3s-worker.service
-   root@k3sworker:~# journalctl -fu k3s-worker.service
```

6. Verifica della corretta installazione, configurazione ed esecuzione dell'orchestratore sul nodo, nonché formazione completa del cluster in seguito al join del Worker Node:

```
-   root@k3sworker:~# tail -f k3sworker.log
-   root@k3sworker:~# k3s check-config
-   root@k3smaster:~# k3s kubectl get nodes -o wide
-   root@k3smaster:~# watch -c -n 0.5 k3s kubectl get pods --all-namespaces –o wide
```