Sardegna FESR 2014/2020 - ASSE PRIORITARIO I

“RICERCA SCIENTIFICA, SVILUPPO TECNOLOGICO E INNOVAZIONE”

Azione 1.1.4 Sostegno alle attività collaborative di R&S per lo sviluppo di nuove tecnologie sostenibili, di nuovi prodotti e servizi

PROGETTI CLUSTER TOP DOWN
R.2.1 – Architettura del VPP

VIRTUALENERGY

Sommario

1. Introduzione 3
2. Infrastruttura IT 4
   2.1 Componenti lato server 4
   2.2 Componenti lato client 6
3. Architettura del VPP 7
   3.1 OSGi 8
   3.2 PowerMatcher 11
   3.3 Energy Flexiblity Platform Interface 17
   3.4 Energy Flexiblity Interface 22

Bibliografia 27

1. Introduzione

Il progetto VIRTUALENERGY è uno dei tanti progetti collaborativi promossi da Sardegna Ricerche attraverso il programma “Azioni Cluster Top-Down”, finanziato grazie al fondo di sviluppo regionale POR-FESR Sardegna 2014-2020. La finalità principale che accomuna queste iniziative è quella di svolgere attività di trasferimento tecnologico nei confronti delle aziende partecipanti, attraverso la risoluzione di problematiche condivise, mediante l’applicazione di competenze e risultati frutto dell’attività di ricerca.

L’obiettivo del progetto VIRTUALENERGY è quello di implementare un prototipo di *Virtual Power Plant* (VPP), ovvero una centrale elettrica virtuale, costituita dall’integrazione e coordinamento di sistemi eterogenei (potenza, controllo e comunicazione) in grado di interfacciarsi con la rete elettrica ed i vari stakeholders che operano sul mercato elettrico per il dispacciamento di energia e la fornitura di servizi ancillari.

Questo documento descrive l’architettura del sistema che, nel suo complesso di componenti hardware e software, consente di realizzare il prototipo di VPP studiato nell’ambito del progetto VIRTUALENERGY.

I vari componenti che concorrono alla realizzazione dell’infrastruttura IT oggetto di questa presentazione sono tutti sistemi di terze parti, selezionati nell’ambito dell’attività di analisi dello stato dell’arte e scouting tecnologico svolta nel corso delle prime fasi del progetto. I criteri utilizzati in questo processo di selezione hanno avuto come obiettivo principale quello di fornire alle aziende del cluster un set di tecnologie a standard aperto, sufficientemente stabili e mature, attraverso cui i soggetti beneficiari dell’intervento di trasferimento tecnologico potessero concretamente andare a pianificare uno sviluppo reale di prodotto o servizio, potendo contare su un’infrastruttura immediatamente disponibile, da intendersi come base di partenza per l’implementazione di applicazioni che avessero come focus principale una gestione flessibile ed efficiente delle risorse energetiche distribuite.

2. Infrastruttura IT

In questo capitolo è descritta l'infrastruttura IT sviluppata per abilitare il funzionamento del prototipo di VPP oggetto di studio. La rappresentazione grafica in Figura 1 consente di individuare i due principali macro-blocchi che caratterizzano il sistema, ovvero la parte lato server, descritta in dettaglio nella sezione 2.1, e la parte lato client, descritta in dettaglio nella sezione 2.2.

2.1 Componenti lato server

L'infrastruttura IT lato server è costituita da un"cloud server", in grado di erogare le diverse tipologie di servizi necessari per il funzionamento del prototipo di VPP oggetto di studio. Per ottimizzare la gestione delle componenti hardware disponibili, si è scelto di utilizzare l'approccio attualmente seguito nella maggior parte dei data center che offrono servizi in cloud agli utenti, ovvero la virtualizzazione delle risorse fisiche. Oltre ai server, in questo caso si è scelto di virtualizzare anche le componenti di storage e networking.

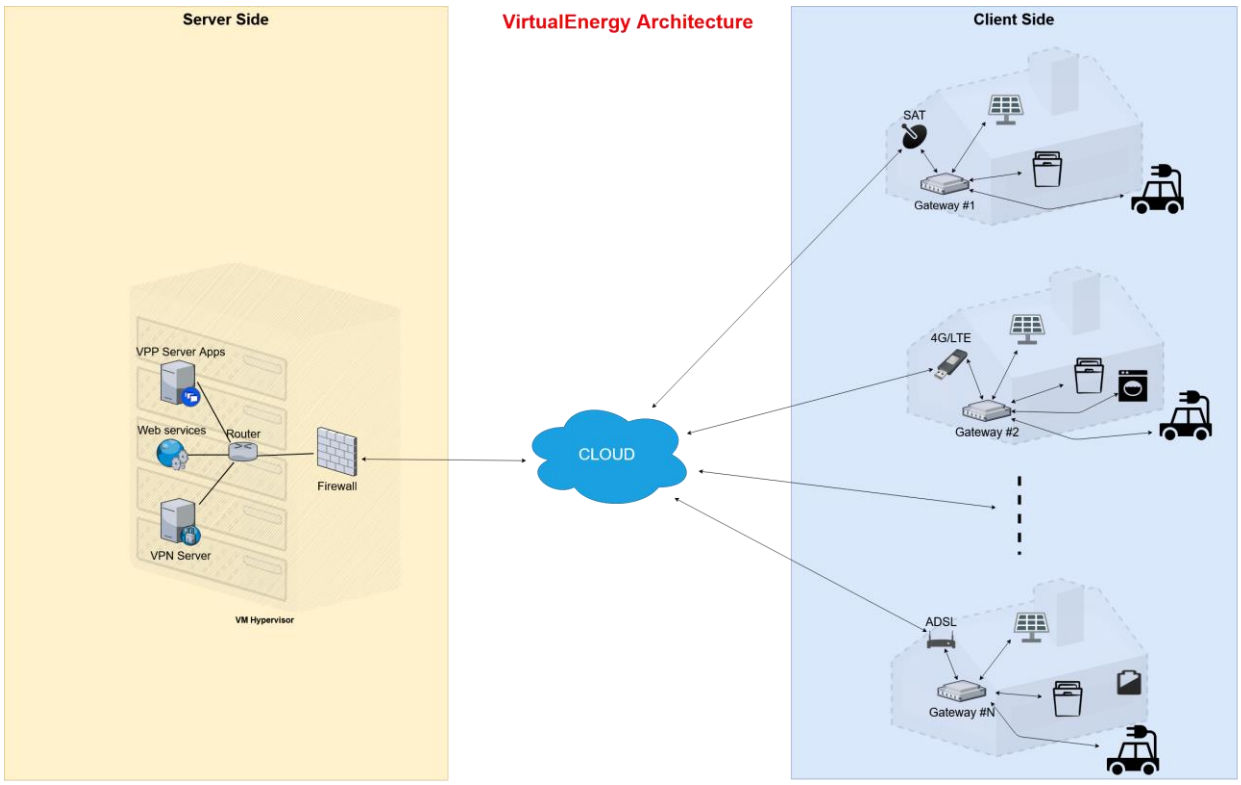

Figura 1: Infrastruttura IT a servizio del VPP

Lo strumento che si è scelto di utilizzare per la virtualizzazione delle risorse è la piattaforma di cloud computing *vSphere*, realizzata dall'azienda VMware [1]. Questa piattaforma è caratterizzata da un certo

insieme di funzionalità, tra cui in particolare un software di virtualizzazione, detto *hypervisor*, che consente di utilizzare in maniera opportuna un certo sottoinsieme delle risorse hardware del server fisico che lo ospita. Per utilizzare questo componente commerciale è necessario attivare una opportuna licenza d'uso, che risulta essere a pagamento se il sistema viene utilizzato all'interno di un ambiente *vSphere*, ma è gratuita per piccole applicazioni in cui sono sufficienti poche macchine virtuali operanti all'interno di un singolo server, e che possono fare a meno delle componenti di *vSphere* che danno continuità di servizio ed elevata affidabilità.

Le componenti principali che caratterizzano questo hypervisor di macchine virtuali, sono in particolare:

- ✓ Un **sistema operativo**, denominato ***ESXi***, costituito da una particolare distribuzione Linux ottimizzata per la virtualizzazione, da installarsi su un supporto di archiviazione collegato direttamente al server fisico o comunque raggiungibile in rete da esso. In questo contesto, il server viene identificato con il termine *host*.
- ✓ Un **microkernel**, denominato ***vmkernel***, che rappresenta l'elemento principale del sistema operativo *ESXi*, e che in particolare implementa tre differenti interfacce: una lato hardware, una lato sistemi *guest*, ed infine una console di servizio.
- ✓ Uno o più **datastore**, ovvero i contenitori logici da considerarsi come un'unità disco, ognuno dotato di un proprio filesystem ad alte prestazioni, denominato ***"Virtual Machine File System"*** (***VMFS***), ottimizzato per memorizzare le macchine virtuali, ognuna delle quali sua volta caratterizzata in particolare da un certo insieme di: immagini disco (file con estensione .vmdk), snapshot, e file di configurazione. Anche questo componente viene installato su un idoneo supporto di archiviazione, collegato direttamente al server fisico o comunque raggiungibile in rete da esso.
- ✓ Una o più **Virtual Machine (VM)**, ognuna corrispondente ad una particolare *Virtual Appliance*, ovvero uno specifico set di servizi offerti dal sistema in cloud, e necessari per il funzionamento del VPP. Una VM è un particolare modulo software che, attraverso un processo di virtualizzazione, crea un ambiente virtuale in grado di emulare il comportamento di una macchina fisica (PC, client o server), grazie all'assegnazione di risorse hardware (porzioni di disco rigido, RAM e risorse di calcolo).

Per implementare le funzionalità di networking necessarie per lo scambio di dati tra i vari elementi del VPP, è stato selezionato un componente denominato **pfsense** ([2]).

Pfsense è una distribuzione software open source, basata su sistema operativo FreeBSD, al momento supportata ufficialmente dall'azienda Netgate, sviluppata con l'obiettivo di fornire un firewall potente, sicuro, completamente configurabile e compatibile con la maggior parte delle piattaforme hardware a basso costo, tra cui i sistemi embedded ed i comuni PC. Il cuore del sistema è rappresentato dal firewall ***PF*** (***Packet Filter***), sviluppato nell'ambito del progetto open source OpenBSD.

Le principali funzionalità che caratterizzano questo modulo software, e che in gran parte sono state utilizzate per l'implementazione dell'infrastruttura di rete a supporto del VPP, risultano essere:

- ✓ **Stateful Firewall**, con controllo granulare e possibilità di funzionare in maniera trasparente al layer 2 (in bridging);
- ✓ **Network Address Translation (NAT)**
- ✓ **High Availability (HA)**, grazie al ***"Common Address Redundancy Protocol" (CARP)***, che permette di configurare due firewall su due macchine identiche per replicarsi e auto-sostituirsi nel caso di guasto di una delle due (il software pfsync si occupa in questo caso di replicare lo stato, la tabella delle connessioni e le regole del firewall, permettendo di passare al secondo dispositivo senza che le connessioni attive di rete cadano).
- ✓ **Load Balancing**, ovvero il bilanciamento del carico di lavoro tra due o più server che si trovano dietro a pfSense
- ✓ **Traffic Shaping,** ovvero l'insieme di operazioni di controllo sul traffico di una rete dati finalizzate a ottimizzare o a garantire le prestazioni di trasmissione, ridurre o controllare i tempi di latenza e sfruttare al meglio la banda disponibile tramite l'accodamento e il ritardo dei pacchetti che soddisfanno determinati criteri
- ✓ **VPN server**, su protocolli IPsec, OpenVPN e L2TP.
- ✓ **PPPoE server**
- ✓ **Grafici Round-Robin-Database (RRD)** ed informazioni sullo stato in tempo reale.
- ✓ **Captive portal**
- ✓ **Gestione uPnP e DNS dinamic**

2.2 Componenti lato client

L'infrastruttura IT lato client è costituita principalmente da due componenti hardware, che caratterizzano il kit di accessori necessario per abilitare il controllo e la la gestione delle risorse energetiche (es. pannelli ed inverter fotovoltaici, sistemi di accumulo, etc…) da parte del VPP:

- ✓ Un dispositivo embedded, da utilizzarsi come gateway, basato su sistema operativo Linux, e dotato almeno di porta USB e interfaccia di rete WiFi, su cui vengono eseguiti i vari moduli software che consentono la comunicazione sia a livello locale, con i dispositivi presenti in campo, sia a livello remoto, con il server in cloud.
- ✓ Un modem 4G/LTE, da connettere al dispositivo gateway per abilitare una connessione ad Internet su rete mobile, eventualmente alternativa alla connessione già disponibile in campo.

In ogni singolo nodo dell'installazione, è possibile aggiungere altri eventuali apparati fisici (sensori ed attuatori), sulla base della particolare applicazione di gestione energetica che si sceglie di implementare:

3. Architettura del VPP

In questo capitolo sono descritti i principali componenti applicativi che caratterizzano a livello funzionale il VPP oggetto di studio. Anche per implementare le funzionalità di controllo e gestione del prototipo di VPP sviluppato nell'ambito di questo progetto, come già osservato del documento R.1.1, nell'ottica di massimizzare il trasferimento tecnologico nei confronti delle aziende aderenti al cluster, si è ritenuto opportuno far riferimento a tecnologie e soluzioni open source già sviluppate e testate in contesti applicativi reali, e per questo ritenute idonee come fonti tecniche da cui partire.

La soluzione individuata, illustrata graficamente in Figura 2, è un progetto software sviluppato dalla *Flexiblepower Alliance Network* (FAN), un'iniziativa senza scopo di lucro nata in Olanda nel 2013 con l'obiettivo di realizzare un insieme universalmente applicabile di standard per la gestione intelligente dell'energia. Il framework sviluppato da questa associazione garantisce l'interoperabilità tra dispositivi intelligenti e smart grid, disponendo inoltre di interfacce flessibili che rendono possibili anche integrazioni future con ulteriori componenti. Con la creazione di una piattaforma attraverso la quale tutti i dispositivi, attuali e futuri, possano comunicare efficacemente con le applicazioni di gestione intelligente dell'energia, la FAN sta cercando di accelerare i progressi verso la realizzazione di soluzioni efficaci. La scelta di rendere disponibili gratuitamente questi strumenti, attraverso una licenza open source, è finalizzata a promuovere il più possibile la diffusione di questo standard. La soluzione denominata ***PowerMatcherSuite***, basata sul framework **OSGi**, descritto nella sezione 3.1, comprende tre componenti: ***PowerMatcher***, ***EF-Pi*** e ***EFI***.

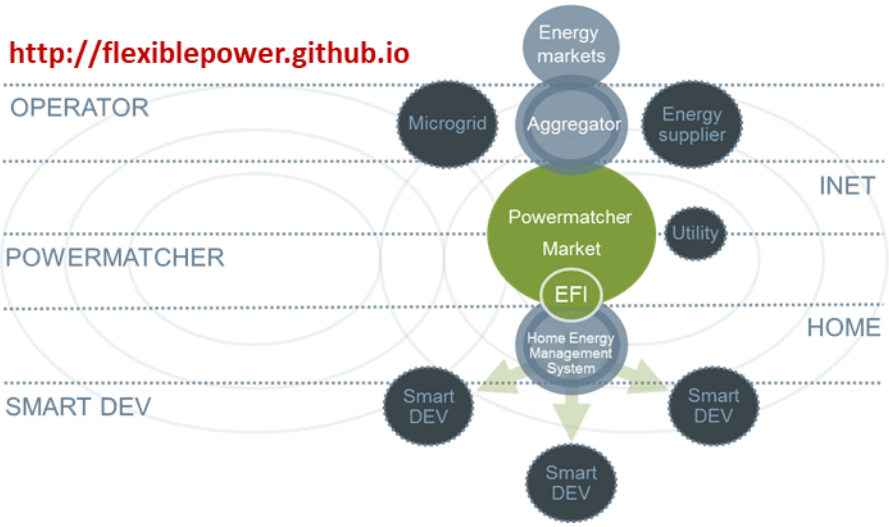

Figura 2: PowerMatcher Suite (FONTE: [3])

PowerMatcher, analizzato in dettaglio nella sezione 3.2, è un sistema che consente di implementare meccanismi di coordinamento delle transazioni all'interno di una smart grid, come ad esempio il bilanciamento tra risorse di energia distribuite e carichi flessibili.

Energy Flexiblity Platform Interface, in breve EF-Pi, analizzata in dettaglio nella sezione 3.3, è invece la piattaforma che consente alle apparecchiature, alla rete ed ai servizi intelligenti di comunicare tra loro.

Energy Flexiblity Interface, in breve EFI, analizzata in dettaglio nella sezione 3.4, è infine il protocollo di comunicazione su cu si basa questa piattaforma. Il protocollo è completamente aperto e disponibile gratuitamente, pertanto qualunque produttore di apparati come ad esempio lavatrici, frigoriferi, unità di climatizzazione, pannelli solari, stazioni per la ricarica di veicoli elettrici, può facilmente rendere i propri dispositivi compatibili con questo sistema.

Le tecnologie sono tra loro complementari, ma possono anche funzionare da sole.

3.1 OSGi

OSGi è l'acronimo di **Open Services Gateway initiative**, ed è una piattaforma di microservizi per il linguaggio di programmazione Java, inizialmente progettato per dispositivi embedded con funzionalità di gateway, come ad esempio: modem, bridge, decoder, TV Box, etc…, ma successivamente ampliato nelle funzionalità in modo da poter essere utilizzato anche in contesti più ampi.

OSGi viene eseguito all'interno di una Java Virtual Machine, e questo implica che lato sistema operativo un certo insieme di servizi risulta essere associato ad un unico processo, ovvero OSGi non risulta essere di per sé un componente distribuito, ma è tuttavia possibile realizzare applicazioni distribuite attraverso questo framework. Altro aspetto che contraddistingue il componente OSGi nell'ecosistema Java, è il fatto di consentire l'installazione e la rimozione di codice al runtime.

OSGi è solamente una specifica, con relativa documentazione, e relative interfacce Java e classi di dati. Il consorzio OSGi tuttavia, non ha creato alcuna implementazione. Pertanto, per poter utilizzare questo framework, è necessario dotarsi di una delle diverse implementazioni open source disponibili, come ad esempio **Apache Felix**, che è anche quella utilizzata in PowerMatcher, oppure Equinox, componente su cui basa il proprio funzionamento l'ambiente di sviluppo integrato **Eclipse**.

La disponibilità di tante diverse implementazioni del framework OSGi, sia di tipo open source che commerciale, ha reso particolarmente complicata la realizzazione di una documentazione di riferimento adeguata, dal momento che ogni particolare implementazione porta con sé specifiche caratteristiche che la contraddistinguono rispetto alle altre.

In Figura 3, è rappresentata l'architettura di una tipica applicazione basata su OSGi.

OSGI ha svariati moduli, che implementano un certo insieme di funzionalità, che si prestano ad essere utilizzati in un ampio spettro di applicazioni. In particolare, le funzionalità utilizzate in PowerMatcher e EF-Pi, risultano essere:

- ✓ Service registry;
- ✓ Declarative services;
- ✓ Metatype;
- ✓ Configuration admin;
- ✓ http whiteboard;
- ✓ Web console

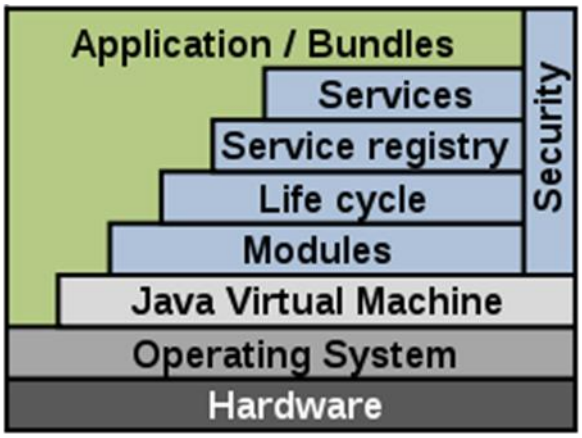

Figura 3: Architettura framework OSGi (Fonte: [4])

Qualsiasi servizio OSGI, è costituito da: una propria *service API*, che descrive come tale servizio può essere utilizzato; un *service consumer*, ovvero la porzione di codice che fa uso del servizio, ovvero richiama l'interfaccia Java del servizio in questione; ed ovviamente una *service implementation*, ovvero l'implementazione delle funzionalità descritte dalla service API.

In runtime, il *service consumer* chiaramente farà uso della *service implementation*. Il grosso vantaggio di questo approccio è la possibilità di utilizzare una differente implementazione del servizio a seconda del contesto o della necessità.

A monte dell'architettura OSGi è possibile eseguire i cosiddetti *bundles*, ovvero dei packages di codice compilato in Java, analoghi ai file jar che si utilizzano normalmente in Java. Un bundle è in pratica uno speciale tipo di file jar. Un bundle è un jar, ma un jar non è detto che sia un bundle. OSGi infatti utilizza informazioni aggiuntive che riguardano un bundle, e che si trovano memorizzate all'interno del file META-INF/MANIFEST.MF presente nell'archivio jar. Un file jar standard non contiene questo tipo di file al suo interno, che sono invece necessari affinché OSGi possa eseguire il bundle. Un aspetto interessante è che

un bundle può comunque essere eseguito come un normale file jar qualora non fosse necessario l'utilizzo di OSGi; questo può tornare utile ad esempio quando si intende utilizzare una libreria senza necessariamente fare uso delle sue funzionalità disponibili attraverso OSGi.

Per poter utilizzare OSGI, oltre ad ovviamente un runtime Java, occorre un'implementazione di OSGI ed un'applicazione, realizzata attraverso un certo insieme di bundles. Un bundle, essendo di fatto una libreria, può contenere nessuno, uno o più servizi.

Lo strumento che si è scelto di utilizzare per lo sviluppo di questi servizi è l'IDE open source Eclipse con i relativi bundle-tools.

BND è un toolbok che automatizza una serie di task per lo sviluppatore di servizi OSGi, come ad esempio la scrittura del file MANIFEST. BND non è uno strumento necessario, ma semplifica notevolmente la vita allo sviluppatore. Esistono naturalmente anche altri tools alternativi a BND per lo sviluppo con OSGi. Il vantaggio di BND è l'utilizzo delle *Java annotations*, ovvero definizioni e direttive inserite direttamente all'interno del codice in corrispondenza del contesto in cui vengono utilizzate, al posto dei files di configurazione, che altrimenti bisognerebbe preoccuparsi di mantenere sincronizzati con il codice. **BNDTools** è il plugin di Eclipse per BND, che offre anche una serie di altre semplificazioni per l'esecuzione di applicazioni OSGi.

Eclipse utilizza l'implementazione OSGi di Apache Felix. Uno strumento utile per gestire tutti i meccanismi interni di OSGI in questo caso è la **Felix web console**. Una volta avviata l'applicazione, l'accesso alla web console a livello locale avviene attraverso l'url http://localhost:8080/system/console/ . Le credenziali di default sono:

- username: admin
- password: admin.

Uno degli aspetti principali di OSGi è il *service registry*. Ogni ambiente OSGi ha un suo registro di servizi, che in pratica è una mappa che collega tutte le Service API, ovvero le classi di interfaccia, agli oggetti che implementano i vari servizi. L'implementazione del servizio è l'elemento di cui ha bisogno l'applicazione per partire. Una volta avviata l'applicazione, risulterà creato un oggetto della service class implementata, e sarà possibile gestire quest'ultimo attraverso il *service registry*, andando a registrare l'oggetto all'interno di questa struttura. L'oggetto che funge da gateway in OSGi per questo tipo di funzionalità è il *bundleContext*. L'elenco dei servizi attualmente registrati nell'applicazione è visualizzabile a livello locale attraverso la web console, all'url http://localhost:8080/system/console/services .

Un altro concetto che è importante conoscere è quelle di *component*, ovvero una parte del modello dichiarativo per i servizi in OSGi. Un component è una speciale tipologia di servizio. Un component è sempre un servizio, ma non sempre un servizio è un component. Un component è utile in quanto risulta essere un elemento configurabile, che può avere una sola istanza (singleton), oppure istanze multiple, e

che presenta aspetti specifici che caratterizzano il suo ciclo di vita. Ad esempio, un component può essere attivato o disattivato; in certi casi può essere utile attivarlo immediatamente all'avvio dell'applicazione, mentre in altri casi è necessario attivarlo solamente quando altri servizi ne richiedono l'attivazione.

In PowerMatcher e EF-Pi, molti elementi sono dei component. La lista dei component ed il loro ciclo di vita in un certo ambiente OSGI è disponibile in locale attraverso la web console all'url: http://localhost:8080/system/console/components .

Metatype è un modulo che consente di esprimere in che modo i component possono essere configurati. Esso contiene tutto quello che occorre sapere per generare il form di configurazione. E' in grado di supportare i tipi di base di Java e le liste.

La *configuration admin* è in sostanza un database di configurazioni, che memorizza la configurazione dei vari componenti in maniera persistente all'interno di un file di testo. Questo elemento si occupa di sincronizzare automaticamente la configurazione con lo stato dell'applicazione. Quando viene creata una nuova *configuration admin*, essa si preoccupa di creare una nuova istanza per ogni componente; allo stesso modo, quando questa configurazione viene distrutta, automaticamente vengono distrutte, dopo essere state disattivate, tutte le istanze dei componenti create con essa. All'avvio dell'applicazione, questa particolare configurazione si preoccupa di caricare tutte le configurazioni memorizzate ed istanziare i relativi componenti. Dal punto di vista del componente, in pratica l'applicazione riprende dal punto in cui era stata interrotta. Per gestire l'inserimento o la rimozione di una o più configurazioni, e di conseguenza l'attivazione o disattivazione di nuovi componenti, un modo assai utile è quello di utilizzare la web console attraverso l'url http://localhost:8080/system/console/configMgr. Esiste anche un altro meccanismo basato sull'utilizzo di file JSON.

L'esecuzione dell'applicazione OSGi avviene attraverso il file bndrun. Il *run requirement* è la collezione di bundles che è necessario avere per poter eseguire correttamente l'applicazione. BNDTools è in grado di risolvere in maniera automatica il problema delle dipendenze di tutti i bundles.

3.2 PowerMatcher

PowerMatcher è una tecnologia che, attraverso l'utilizzo di particolari componenti software chiamati agenti, sfrutta i principi di base dell'economia che regolano il funzionamento dei mercati per aggregare e dispacciare la flessibilità nella produzione e/o nel consumo di energia che le risorse energetiche sono in grado di fornire. L'idea di fondo, rappresentata graficamente in Figura 4, è quella di avere una struttura ad albero, con un certo numero di ***agenti dispositivo***, ognuno dei quali rappresenta un differente apparato in grado di produrre o consumare energia, come ad esempio: un sistema di riscaldamento, una stazione per la ricarica di veicoli elettrici, un veicolo elettrico, un impianto fotovoltaico, e così via. Ogni singolo apparato è rappresentato in PowerMatcher attraverso il suo agente corrispondente, che è in grado di inviare delle

offerte e ricevere dei prezzi sulla base del comportamento che implementa l'apparato. Il dispositivo agente è il livello che consente di interfacciare un certo apparato con i principi economici, essendo in grado di interloquire con entrambi i sistemi. Altri elementi che si trovano in PowerMatcher sono i cosiddetti ***agenti concentratore***, che aggregano le varie offerte che arrivano loro dai diversi agenti dispositivo ad essi collegati, comportandosi a loro volta come livello di interfaccia tra i dispositivi agente e l'***agente banditore***, che si trova al livello superiore, inviando a quest'ultimo delle offerte e ricevendo da esso dei prezzi. Gli agenti banditore sono quelli a cui spetta il compito di realizzare le aste di mercato. L'agente banditore interagisce infine con il livello più alto, in cui si trova l'***agente obiettivo***, che implementa la logica di business desiderata, ovvero l'obiettivo di ottimizzazione che l'intero cluster di PowerMatcher concorrerà a raggiungere. L'implementazione di PowerMatcher considerata, è in grado di interagire con i dispositivi attraverso il componente EFI, che operativamente realizza l'interfaccia di comunicazione tra gli apparati e i dispositivi agente.

Come già osservato, in PowerMatcher la topologia della rete di agenti è quella ad albero, in cui le foglie sono rappresentate dai dispositivi, ovvero apparati di varia natura, come ad esempio sistemi di ricarica per veicoli elettrici, pannelli solari, batterie, etc… PowerMatcher è un meccanismo basato su logiche di mercato, in cui ogni dispositivo agente interagisce con un certo agente concentratore, effettuando ad esso delle offerte e ricevendo da esso dei prezzi. L'agente concentratore aggrega tra loro tutte le offerte che riceve dai vari agenti dispositivo a cui risulta connesso, per poi effettuare un'offerta al nodo radice, rappresentato da un agente banditore. Nei confronti degli agenti dispositivo quindi, l'agente concentratore assume il ruolo di operatore di mercato, mentre rispetto all'agente banditore assume lo stesso ruolo dell'agente dispositivo, svolgendo di fatto il ruolo di aggregatore dei dispositivi rappresentati dai vari agenti ad esso connessi. L'agente banditore riceve tutte le offerte da parte degli altri agenti concentratori, a cui propone delle offerte distribuendo i prezzi tra i vari cluster, che a loro volta vengono ridistribuiti singolarmente ai livelli sottostanti. La struttura ad albero rende PowerMatcher un sistema altamente scalabile, in quanto è possibile rendere trasparente ai livelli superiori la complessità di un certo agente concentratore, che di fatto agisce come un singolo agente. Ogni cluster ha un solo agente banditore, che ha il compito di mantenere l'intero cluster bilanciato, ovvero rendere la produzione totale generata quest'ultimo uguale al suo consumo totale. Esiste infine un agente speciale, denominato agente obiettivo, che diventa necessario nel caso in cui PowerMatcher debba essere utilizzato in configurazione VPP. In questo caso, tutti i vari cluster devono concorrere al raggiungimento di una certa funzione obiettivo, come ad esempio l'interfacciamento con un certo mercato elettrico, oppure la fornitura del servizio di bilanciamento, che viene implementata proprio dall'agente obiettivo, il quale si comporterà esattamente come gli altri agenti, cercando di "pilotare" il mercato in modo tale che il comportamento del cluster sia tale da soddisfare l'obiettivo atteso.

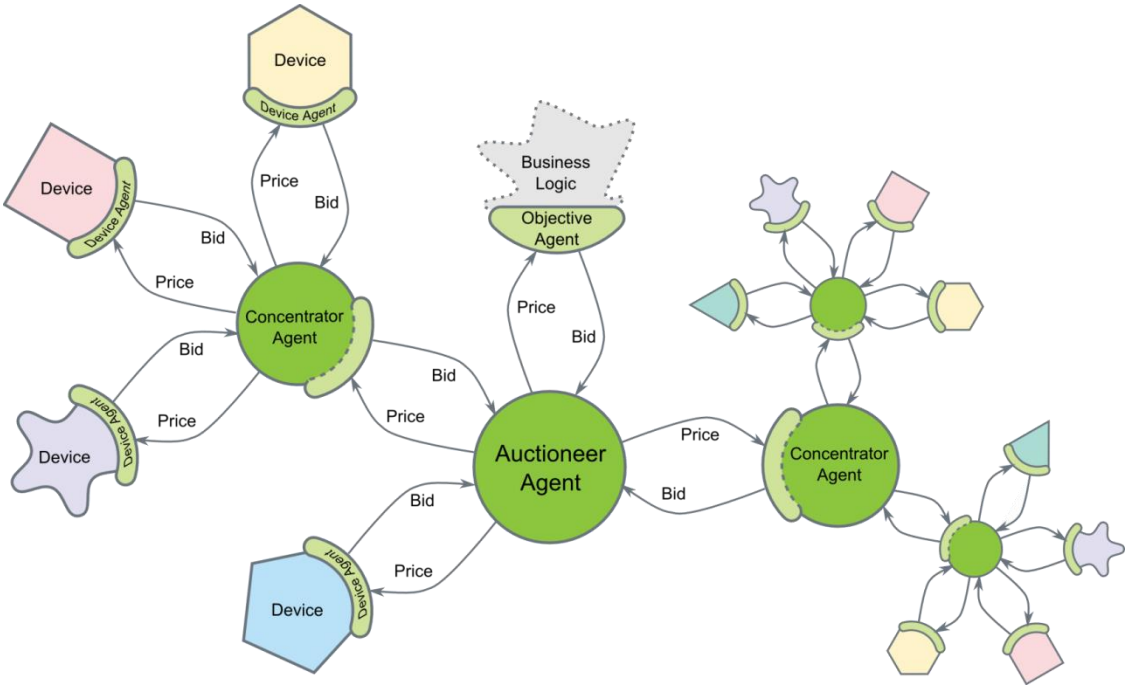

Figura 4: Architettura PowerMatcher (Fonte: [5])

Come rappresentato in Figura 5, in PowerMatcher gli agenti possono avere uno o due ruoli: quello di ***Agent*** e quello di ***Matcher***. L'Agent propone delle offerte al suo Matcher, sulla base della flessibilità di produzione e/o consumo che rappresenta. Il Matcher riceve le offerte da parte dei vari Agent e fornisce loro un prezzo sulla base delle offerte ricevute. In genere ogni Agent è associato ad uno ed un solo Matcher. I due ruoli appena descritti possono essere assunti da differenti tipologie di agente. In una gerarchia PowerMatcher i nodi foglia sono come già detto gli agenti dispositivo, che possono assumere esclusivamente il ruolo di Agent, mentre il nodo radice è l'agente banditore, che in genere assume il ruolo di Matcher.I nodi intermedi sono gli agenti concentratore, che assumono sempre entrambi i ruoli di Agent e Matcher.

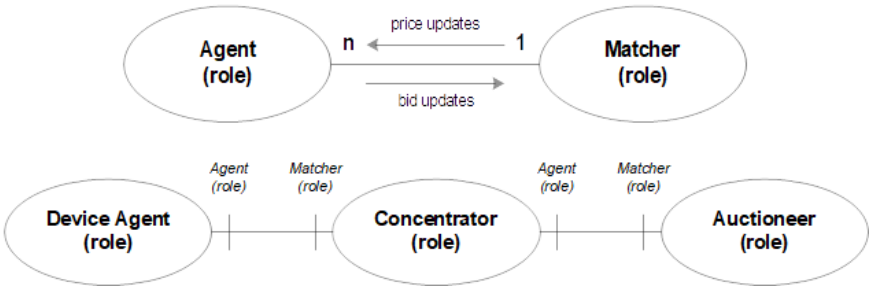

Figura 5: Ruoli in PowerMatcher (Fonte: [5])

Le componenti fondamentali di PowerMatcher sono offerte e prezzi, che vengono scambiati in tempo reale in accordo con le dinamiche tipiche di un mercato economico. Un'offerta esprime la disponibilità istantanea a consumare o produrre energia in maniera flessibile. Un prezzo esprime il valore istantaneo associato a domanda o offerta. In PowerMatcher la regola è quella di considerare validi un'offerta o un prezzo fino all'aggiornamento dell'offerta o del prezzo successivo. Trattandosi di un mercato in tempo reale, gli agenti sono liberi di aggiornare i loro prezzi o le loro offerte in qualsiasi momento, sulla base dell'evoluzione istantanea che essi registrano.

In conclusione, PowerMatcher è un'infrastruttura che, attraverso l'interazione di agenti che attuano semplici logiche di controllo a livello locale, è in grado di agire a livello di sistema portando quest'ultimo ad assumere comportamenti o raggiungere obiettivi complessi. Attraverso PowerMatcher è possibile raggiungere obiettivi multipli a livello di sistema, come ad esempio il bilanciamento e la gestione della congestione in una rete elettrica. I sistemi multi-agente ed i mercati elettronici forniscono gli strumenti per realizzare sistemi distribuiti in grado di eseguire azioni in contesti ad elevata complessità, come ad esempio la gestione dell'infrastruttura di rete.

Il codice sorgente di PowerMarcher può essere scaricato dall'url https://github.com/flexiblepower/PowerMatcher . Questo repository contiene in particolare il framework PowerMatcher e consente di implementare un progetto basato su PowerMatcher, ma non è tuttavia un'applicazione pronta all'uso, in quanto per realizzare l'applicazione desiderata è necessario implementare la logica che dovranno attuare l'agente obiettivo ed i vari agenti dispositivo al loro interno. Fortunatamente, il framework fornisce svariati esempi che possono essere presi come riferimento da cui partire per implementare le logiche proprie dell'applicazione che si vuole realizzare.

La logica centrale implementata attraverso PowerMatcher ed il framework EF-Pi, descritto nella sezione 3.3, forniscono un grosso aiuto allo sviluppatore, ma tuttavia entrambi non fanno parte del core di PowerMatcher.

Sia PowerMatcher che EF-Pi utilizzano entrambi un elemento chiamato *flexiblePowerContext*, che è possibile trovare all'interno di ogni singolo agente dispositivo. In particolare, si tratta di un'interfaccia definita nella API flexiblepower, condivisa tra PowerMatcher e EF-Pi, ed utilizzata come servizio OSGi, che è in grado di eseguire alcune operazioni utili come ad esempio fornire l'orario attuale, eseguire in maniera asincrona o programmare l'esecuzione di task, tra cui l'aggiornamento dei modelli o l'interrogazione periodica dei dispositivi per l'acquisizione di nuovi dati. Il motivo per cui si utilizza questo componente è la possibilità di utilizzare per esso diverse implementazioni. Questo aspetto torna molto utile ad esempio quando si ha la necessità di realizzare simulazioni che differiscono tra loro per orario diverso rispetto a quello attuale, o per la velocità di esecuzione. Un'altra caratteristica utile che presenta la API flexiblepower è la disponibilità di classi per gestire unità di misura codificate nel Sistema Internazionale (SI) ed in altri standard metrici di uso comune.

Come già osservato, un importante concetto presente in PowerMatcher è quello aggregato, o cluster. Ogni agente appartiene ad un cluster PowerMatcher, identificato dal suo *ClusterID*. Agenti appartenenti a cluster differenti non saranno in grado di comunicare tra di loro. Ogni cluster ha un *Marketbasis*, che consiste in pratica in una configurazione del mercato. Un *Marketbasis* in particolare definisce per un certo cluster: la *commodity*, che di default è l'elettricità, la *valuta*, che di default è l'euro, il *prezzo minimo* e *massimo di mercato* per l'energia, che di default sono fissati rispettivamente pari a 0 e 1, la *risoluzione* delle offerte, ovvero il numero di incrementi di prezzo tra valore minimo e massimo, che di default è pari a 100. Questo parametro in genere influenza anche la complessità computazionale, in quanto un valore elevato corrisponderà tipicamente ad un maggior numero di dati da gestire, così come viceversa un valore più piccolo sarà associato a condizioni meno complesse di cui il sistema dovrà tener conto. La configurazione del *Marketbasis* viene ricevuta direttamente da un agente non appena questo si connette al suo agente matcher. L'agente banditore definisce il *Marketbasis*. Gli agenti concentratori hanno il compito di distribuire questa configurazione agli altri agenti con cui si connettono.

Ogni agente in PowerMatcher è un componente OSGi. Gli agenti in PowerMatcher sono pertanto configurati attraverso strumenti come *metatype* e *configuration admin*, presentati nella sezione 3.1. Ogni agente PowerMatcher ha impostate almeno due proprietà, che sono *AgentID* e *ClusterID*.

Per poter interagire tra loro, i vari agenti devono necessariamente registrarsi nel *service registry*, implementando apposite interfacce Java. Gli agenti dispositivo si registrano attraverso l'interfaccia *AgentEndpoint*, gli agenti banditore si registrano invece attraverso l'interfaccia *MatcherEndpoint*, mentre gli agenti concentratore si registrano sia come *AgentEndpoint* che come *MatcherEndpoint*. Il modo in cui gli agenti risultano connessi dipende dal loro *AgentID*. In particolare, gli *AgentEndpoint* devono aver configurata anche la proprietà *desiredParentID*, ovvero l'*AgentID* dell'agente a cui risultano connessi. Un altro modulo di PowerMatcher, chiamato *runtime*, tiene traccia di tutte le configurazioni ed automaticamente si occupa di connettere tra loro gli agenti. Pertanto, non ci si deve preoccupare di connettere tra di loro i vari agenti. Tecnicamente, ogni volta che il PowerMatcher *runtime* connette tra loro due agenti, crea un oggetto *Session*, che è in pratica un oggetto che funge da intermediario tra i due canali di comunicazione. Questo implica quindi che, operativamente, ogni volta che un agente pubblica un'offerta, esso in realtà sta inviando questa offerta alla sessione, la quale si preoccuperà di inoltrarla all'agente banditore. Allo stesso modo, ogni volta che un agente banditore pubblica un prezzo, in realtà sta inviando questo alla sessione, che si preoccuperà di inoltrarlo all'agente dispositivo. Per ogni nuovo invio, l'oggetto session genera un nuovo task che viene messo nel *thread pool* per evitare potenziali stalli. Nelle simulazioni è possibile utilizzare differenti implementazioni dell'oggetto session, in modo da gestire in maniera differente il flusso di offerte e prezzi, ossia dei segnali di *Bidupdate* e *PriceUpdate*.

Le offerte vengono in genere memorizzate internamente in degli array. In un *Marketplace* configurato con i valori di default pertanto ad esempio, ogni array di offerte risulterà costituito da 100 elementi, in cui il primo

elemento corrisponderà al prezzo minimo offerto, espresso come quantità minima di potenza promessa, indicata in Watt, mentre l'ultimo elemento sarà il prezzo massimo offerto, ovvero la quantità massima di potenza promessa. Tuttavia, esiste una modalità alternativa di costruire le offerte, che risulta essere molto più comoda quando si sta implementando una certa strategia di offerta. In genere, è più facile creare un *PointBidBuilder*, ovvero una classe attraverso cui è necessario definire solamente le coordinate di alcuni punti nell'offerta, ed automaticamente il *PointBidBuilder* genererà l'array di offerte corrispondente andando a riempirlo con i valori ottenuti interpolando linearmente i punti definiti in precedenza, estendendo il primo e ultimo punto forniti, rispettivamente fino al primo ed ultimo prezzo dell'array.

PowerMatcher, come già osservato, è un framework altamente scalabile grazie al concetto degli agenti concentratori. Se infatti ad esempio ogni concentratore ha 100 agenti connessi ad esso, con 3 soli livelli di concentratori è possibile supportare un milione di agenti dispositivo. PowerMatcher, è comunque un sistema basato su eventi, pertanto ogni segnale di *BitUpdate* ricevuto da un concentratore deve inoltre avere un corrispondente *BitUpdate* inviato dal concentratore al suo agente al livello superiore. Senza un meccanismo che consenta di ridurre il numero di segnali di aggiornamento delle offerte, il banditore verrebbe subissato dalle troppe richieste che troverebbe a ricevere dagli agenti ai livelli inferiori. Per limitare questo problema, ovvero ridurre il numero di offerte, i concentratori possono aggregare tra loro le offerte che ricevono dai tutti gli agenti connessi, ed inviare questo aggregato al livello superiore al posto delle singole offerte. Questo meccanismo viene implementato con una cache in cui vengono mantenuti i segnali di *BidUpdate* ricevuti entro un certo intervallo di tempo, definito mediante il parametro *minTimeBetweenBidUpdates*, il cui valore di default è pari a 1000 millisecondi. Questo significa operativamente che, se in un secondo il concentratore riceve più di una richiesta di offerta, esso invierà al livello superiore un solo segnale di *BidUpdate*, che aggregherà al suo interno le varie offerte ricevute dai livelli inferiori. L'effetto collaterale di questa soluzione è quello di introdurre un piccolo delay nella comunicazione.

Per analizzare il comportamento di un certo cluster PowerMatcher, spesso si ha l'esigenza di sapere cosa sta succedendo, ovvero che tipo di prezzi ed offerte stanno generando i vari agenti in esso presenti. La maggior parte degli agenti è in grado di registrare il proprio comportamento, attraverso un'implementazione del servizio *ObservableAgent*. Per questi agenti è possibile implementare una o più istanze di *AgentObserver*, ovvero agenti osservatori. Un *AgentObserver* è una classe che mantiene le informazioni relative agli *AgentEvent*. Esistono fondamentalmente tre diverse tipologie di eventi, che sono in particolare: 1) l'evento *AggregatedBidUpdateEvent*, che si verifica quando un agente ha delle offerte aggregate; 2) l'evento *BidUpdateEvent*, che si verifica quando un agente invia o riceve un'offerta; 3) l'evento *PriceUpdateEvent*, che si verifica quando un agente riceve un prezzo. Attraverso gli *AgentObserver* è' possibile quindi collegarsi ad uno o più agenti e mantenere traccia del loro comportamento andando a

scrivere su un database o su un file di testo i vari eventi di invio o ricezione di offerte e prezzi che essi generano.

Per le applicazioni in campo, come quella considerata in questo studio, si ha necessità di avere un cluster distribuito. In questo caso, è stato previsto che gli agenti banditore si trovino in esecuzione presso il server centrale; mentre gli agenti dispositivo vengano eseguiti ognuno in un apposito computer embedded, costituito ad esempio da un Raspberry PI, nei pressi dell'apparato che esso si troverà a gestire, attraverso l'interfaccia EF-Pi. Questo implica che si avrà un cluster all'interno del quale gli agenti vengono eseguiti all'interno di differenti Java Virtual Machine. Il problema da risolvere in questo caso sarà pertanto quello di connettere tra di loro questi elementi. PowerMatcher lavora attraverso un protocollo di comunicazione basato sull'uso di *websocket* per connettere gli agenti attraverso una rete. Il package che implementa questa funzionalità è il *net.PowerMatcher.remote.websockets*. Sulla macchina che assumerà il ruolo di *Matcher*, si utilizzerà il *PowerMatcherWebSocketServlet*, ovvero una servlet http in grado di ricevere connessioni multiple in ingresso, attraverso cui un certo numero di agenti può connettersi, che registra se stessa come un *AgentEndpoint*. Sulla macchina che assumerà il ruolo di *Agent*, si utilizzerà invece il *PowermarcherWebSocketClient*, ovvero una servlet http in grado di ricevere le offerte in ingresso ed inoltrarle in uscita, che registra sé stessa come un *MatcherAgent*. E' possibile pensare a questa configurazione come se fosse un concentratore distribuito. Il protocollo utilizza il formato JSON per codificare le offerte ed i prezzi.

3.3 Energy Flexiblity Platform Interface

Il componente EF-Pi è una piattaforma software studiata per favorire l'interfacciamento tra i vari dispositivi che caratterizzano una certa applicazione, in modo da rendere queste delle risorse di energia flessibili. Pertanto, EF-Pi non rappresenta lo strumento che si deve utilizzare per sviluppare l'applicazione stessa che realizza la gestione della flessibilità energetica, bensì la piattaforma di cui l'applicazione necessita per poter funzionare. EF-PI fornisce una struttura per implementare *Resource Manager*, *Resource Drivers*, Driver di protocollo, *Energy App*, *Customer Energy Manager*, *Controller*, (questi ultimi tre componenti sono più o meno la stessa cosa), ed anche per monitorare l'applicazione stessa o collezionare misure. Inoltre, offre una interfaccia utente basata su widget, un sistema di comunicazione tra componenti asincrono basato su messaggi, il caricamento delle configurazioni attraverso XML, e consente infine di eseguire simulazioni su una base dei tempi più veloce di quella reale.

Un modo efficace per rendere l'idea di cosa sia effettivamente il componente EF-Pi è quello di fare un parallelo con l'app store del mondo Android. Quando si acquista uno smartphone oramai, la maggior parte delle funzionalità principali non sono quelle che porta con sé il sistema operativo Android. In genere, ogni specifica funzionalità viene resa disponibile attraverso un'app da installarsi a parte. Il componente EF-Pi può essere a tutti gli effetti considerato come una sorta di sistema operativo sopra cui è possibile far girare

applicazioni specifiche, come ad esempio un driver per ogni differente tipo di apparato che si ha necessità di gestire, oppure *Energy Application*, come ad esempio PowerMatcher o altri tipi di *Customer Energy Manager (CEM)*. Queste applicazioni possono a loro volta essere configurate a seconda delle particolari necessità. Ad esempio, per un driver è possibile specificare i parametri di connessione richiesti da un certo protocollo, oppure quale *Energy Application* può controllare uno specifico dispositivo, andando ad impostare una opportuna interfaccia di connessione, come la EFI discussa in sezione 3.4, tra *Energy Application* e driver del dispositivo. Attraverso il canale di comunicazione implementato dalla EFI, i due moduli software, ovvero l'*Energy Application* ed il driver, possono scambiarsi informazioni relative alla flessibilità energetica del dispositivo ed inviare comandi a quest'ultimo. Al livello superiore di EF-Pi si trovano infine un'interfaccia utente web-based ed una sorta di centro di gestione, che consente di controllare in maniera remota le varie istanze dei componenti utilizzati.

I componenti principali della piattaforma sono implementati nel progetto fpai-core, i cui sorgenti sono scaricabili dal repository https://github.com/flexiblepower/fpai-core . Tuttavia, in altri repository del progetto, raggiungibili dall'url https://github.com/flexiblepower , oltre al core, sono disponibili anche altri componenti che è possibile eseguire sopra EF-Pi, quali drivers e *Resource Managers* per alcuni apparati reali, *Resource Managers* che simulano alcuni apparati, componenti per il monitoraggio, ed anche un'applicazione PowerMatcher per EFI, che include quattro agenti dispositivo generici per le quattro categorie EFI discusse nella sezione 3.4.

EF-Pi fornisce fondamentalmente un approccio che non forza lo sviluppatore ad utilizzare una struttura in particolare, ma intende agevolare quest'ultimo portandolo ad utilizzare una convenzione per l'organizzazione dell'applicazione, che tuttavia non è vincolante. EF-Pi viene eseguita sopra il framework OSGi, pertanto è possibile utilizzare le strutture, i componenti e le altre features che questo framework mette a disposizione.

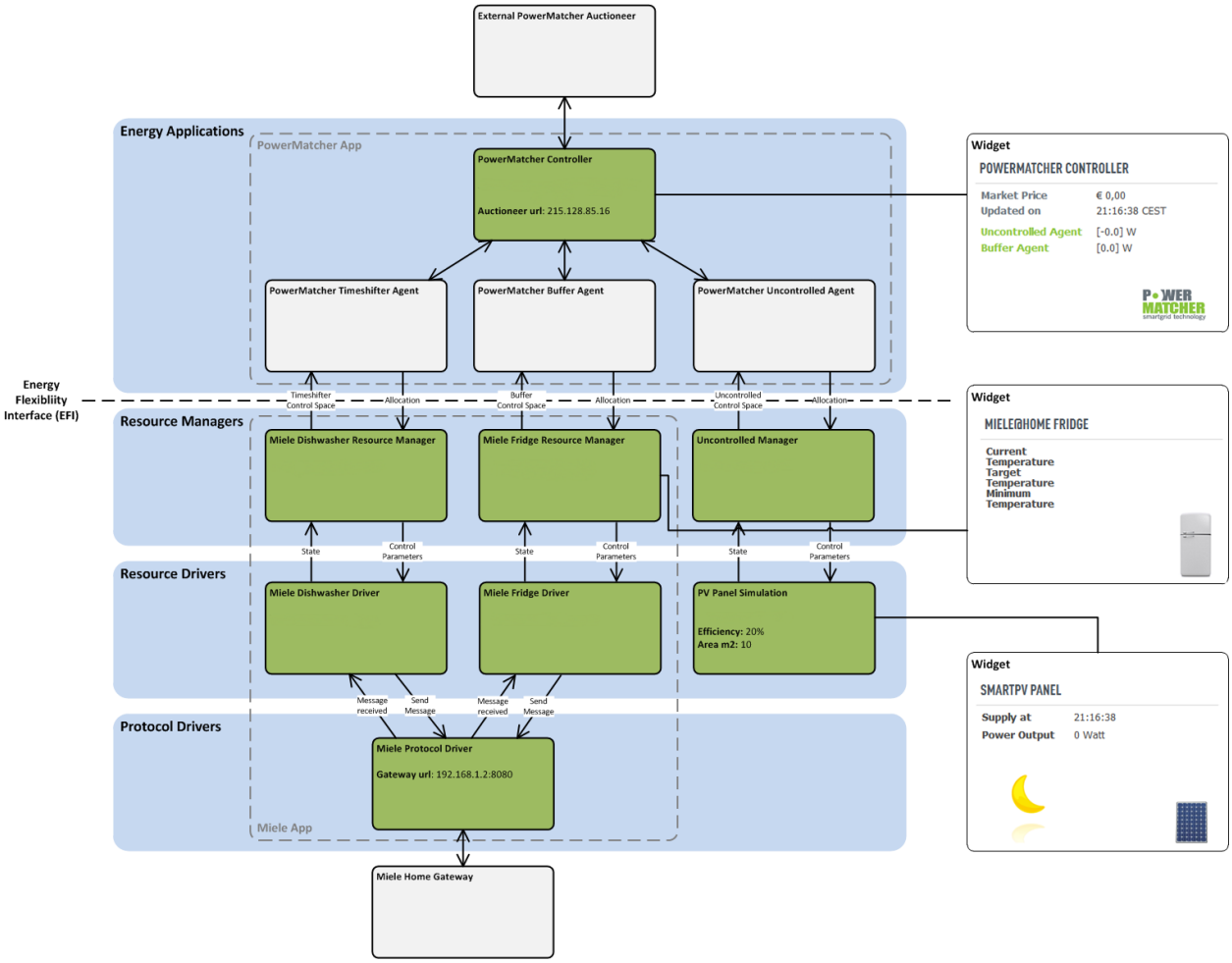

Figura 6: EF-Pi - esempio di configurazione (Fonte: [6])

La Figura 6 , estratta dalla documentazione originale del software, mostra attraverso un esempio alcuni dei componenti che è possibile trovare in EF-Pi. Partendo dal livello superiore, ovvero le *Energy applications*, troviamo la PowerMatcher App, che in questo caso sta controllando una lavastoviglie, un congelatore del produttore Miele, ed un simulatore di impianto fotovoltaico. Notiamo la presenza di un componente OSGi, chiamato PowerMatcher controller, caratterizzato da un proprio widget, che ha creato tre diversi agenti PowerMatcher. Gli agenti, non essendo dei componenti OSGI, sono indicati in color grigio chiaro. Il livello sottostante è quello dei *Resource Managers*, in cui per ogni agente istanziato è presente un *Resource Manager*, ovvero un componente OSGI, che comunica con il relativo agente attraverso messaggi basati sulla EFI. Nel livello inferiore troviamo un *Resource Driver* per ogni *Resource Manager*, il cui obiettivo è acquisire dati dall'apparato e trasferirli al livello superiore. La funzione del *Resource Manager* a sua volta è quella di tradurre i dati grezzi provenienti dall'apparato in dati di tipo EFI, ovvero informazioni inerenti allo stato di flessibilità dell'apparato. La comunicazione tra questi due livelli, tipicamente, avviene attraverso

l'invio di stati da parte del *Resource Driver* al *Resource Manager*, il quale risponde a sua volta con l'invio di parametri di controllo, ovvero le istruzioni che il *Resource Driver* dovrà comunicare a sua volta all'apparato. L'aspetto di questi messaggi dipende fortemente dal tipo di apparato che si sta gestendo. Si può osservare in particolare, come l'app denominata Miele sia costituita da un insieme di cinque bundles, relativi alla gestione dei due elettrodomestici presenti in questo esempio. L'ecosistema Miele ha la particolarità di avere un proprio home gateway, ovvero un'unità di elaborazione che consente di comunicare con gli elettrodomestici su powerline attraverso protocollo TCP/IP. Questo gateway, sostanzialmente, espone gli stati di tutti gli elettrodomestici Miele installati presso l'abitazione in un'interfaccia basata su xml, e sempre attraverso la stessa interfaccia consente di inviare parametri di controllo per modificare il comportamento di questi ultimi. In questo esempio specifico, è presente un solo Protocol Driver che si occupa di scambiare con l'Home Gateway della Miele tutti i messaggi che riceve da entrambi *Resource Drivers* associati ai due elettrodomestici presenti. Il protocollo di comunicazione utilizzato a questo livello per lo scambio delle informazioni è quello basato su xml secondo le specifiche di Miele. Di conseguenza, il *Protocol Driver* è un modulo software che è stato sviluppato appositamente per comunicare con questa marca di elettrodomestici. Non occorre pertanto per esso definire alcuna struttura che riguardi l'interfaccia EFI. Tuttavia, l'approccio più conveniente da seguire in questo caso è quello di definire una service API, e registrare il *Protocol Driver* come componente OSGI nel *service registry* attraverso questa service API, in modo tale che i *Resource Drivers* possano trovare il *Protocol Driver* nel *service registry* ed interagire con esso. Per ogni componente, come visto, sarà possibile definire una certa configurazione. Ad esempio, il PowerMatcher Controller in questo caso particolare risulta essere un banditore raggiungibile attraverso un indirizzo ip. Allo stesso modo, per il pannello fotovoltaico è possibile definirne la relativa efficienza e la sua superficie.

Un CEM basato sul protocollo EFI, può dover implementare la logica di controllo per tutte e quattro le differenti categorie discusse nella sezione 3.4. E' possibile utilizzare l'interfaccia *ControllerManager* per indicare a EF-Pi che il componente è un CEM oppure un'*Energy Application*. Questa interfaccia si chiama *ControllerManager* poiché l'idea è quella di avere un solo controller per dispositivo, mentre un CEM di solito controlla più dispositivi. Per implementare un CEM basato su EFI è necessario implementare un'interfaccia più specifica che si chiama *EfiControllerManager*.

Per sviluppare il proprio *Resource Manager* occorre implementare l'interfaccia *ResourceManager* oppure l'interfaccia specifica per la particolare categoria EFI che si vuole gestire. Se si vogliono gestire in maniera automatica alcuni comandi, è possibile inoltre estendere la classe astratta *AbstractResourceManager*. Questa classe assume che il *Resource Manager* comunicherà con il *ResourceDriver* utilizzando classi di dati che estendono le classi astratte *ResourceState* e *ResourceControlParameters.* Il *Resource Manager* traduce i parametri specifici dell'apparato in informazioni che riguardano il livello di flessibilità secondo lo standard EFI. Questa operazione non è banale, in quanto oltre a richiedere al driver le informazioni inerenti

allo stato di funzionamento dell'apparato, necessità di una conoscenza sul tipo di flessibilità che quest'ultimo è in grado di offrire, e che dovrà essere modellata attraverso opportuni parametri in EFI; inoltre, se l'apparato può essere controllato, richiede l'invio di opportune istruzioni all'apparato.

Per sviluppare il proprio *Resource Driver* occorre implementare l'interfaccia *ResourceDriver*, oppure estendere la classe astratta *AbstractResourceDriver*. Anche in questo caso, la classe assume che il *Resource Driver* comunicherà con il *Resource Manager* utilizzando classi di dati che estendono le classi astratte R*esourceState* e *ResourceControlParameters*. Nei casi in cui si hanno diversi *Resource Drivers* che condividono lo stesso protocollo di interfaccia, come ad esempio ZIgBee, Z-Wave, Modbus, etc…, è possibile implementare un *Protocol Driver* separato attraverso apposito bundle o componente. EF-Pi infatti non fornisce alcuna struttura per fare ciò; pertanto, per integrare questi componenti software, in genere si utilizzano meccanismi come il *service registry* descritto nella sezione 3.1, o il framework di messaggistica.

Il framework di messaggistica rappresenta una parte importante di EF-Pi. Nella maggior parte delle applicazioni relative alle smart grid, si ha la necessità di adottare una programmazione basata su eventi. Questo accade perché in tali applicazioni i componenti hanno bisogno di scambiarsi tra di loro dati in ogni momento, senza l'introduzione dei ritardi che inevitabilmente si avrebbero con l'utilizzo di meccanismi di polling. La presenza di diversi livelli di astrazione inoltre, rende ancora più impraticabile la strada del polling, in quanto ogni livello introdurrebbe un suo ritardo che si andrebbe a sommare a quello degli altri livelli rendendo particolarmente significativo il ritardo complessivo nella comunicazione. Di conseguenza, l'approccio event-driven che si è scelto di utilizzare prevede che, ogni volta che un nuovo dato arriva, il sistema avvia la sua elaborazione e conseguente aggiornamento dei modelli, a cui segue immediatamente la creazione di una nuova offerta ed il suo invio all'altro agente. Tipicamente, si ha necessità di utilizzare il polling unicamente per lo scambio di dati con gli apparati a livello di driver del dispositivo. Dal *Device Driver* ai livelli superiori lo scambio di informazioni avviene unicamente in modalità ad eventi. Per standardizzare questo scambio di dati, si è scelto di utilizzare delle classi di dati per la comunicazione, che principalmente si occupano di memorizzare informazioni ed implementano poca logica di controllo. Per poter comunicare attraverso il framework di messaggistica di EF-Pi, i componenti devono anzitutto registrarsi con il servizio *EndPoint*, che in sostanza è un'interfaccia Java. Ogni componente che si registra come *EndPoint* può definire una o più porte. Per una porta è necessario: definire quali classi di messaggi è possibile ricevere e quali è possibile inviare, assegnare un nome alla porta ed infine definire la sua cardinalità, ovvero quanti canali di comunicazione è possibile aprire attraverso questa porta. Ad esempio, un *Resource Manager* dovrebbe potersi connettere ad un solo controller, mentre il controller può comunicare con più dispositivi. Il framework di comunicazione autorizza solamente le connessioni tra porte compatibili in termini di messaggi che esse possono inviare e ricevere. La gestione di queste connessioni può essere effettuata sia a livello di codice, attraverso il servizio denominato *ConnectionManager*, che è possibile trovare nel *service registry*. Tuttavia, il servizio *ConnectionManager* mette a disposizione anche una sua interfaccia

utente, sia nella dashboard di EF-Pi che nella Web Console Felix, attraverso cui è possibile attivare o disattivare connessioni mediante un intuitivo approccio basato su drag-and-drop. In quest'interfaccia è presente una funzione di "autoconnect" che cerca di connettere in maniera automatica tra loro le varie porte sulla base delle loro caratteristiche di compatibilità.

Un altro elemento importante in EF-Pi è il framework di osservazione. Spesso si ha l'esigenza di monitorare cosa succede all'interno di EF-Pi, per gli stessi motivi che abbiamo visto rendono utile l'*AgentObserver* in PowerMatcher. Un'osservazione è fondamentalmente una struttura dati chiave-valore libera, ovvero una mappa tra un nome di un componente ed un oggetto, che può essere creata da ogni componente in qualsiasi istante di tempo. Ogni osservazione contiene alcuni metadati, come ad esempio l'istante di tempo associato all'osservazione ed il componente che ha generato l'osservazione. *ObservationProvider* è un'interfaccia che viene utilizzata dai componenti che generano osservazioni, mentre l'interfaccia *ObservationConsumer* è l'interfaccia utilizzata dai componenti che ricevono osservazioni. Le modalità attraverso cui è possibile utilizzare queste osservazioni sono molteplici.

Una modalità molto utilizzata per la gestione delle osservazioni è ad esempio quella di scriverle in un'istanza di InfluxDB [7], ovvero un database di serie temporali, particolarmente adatto per memorizzare misure, ed utilizzare il tool Grafana [8] per visualizzare i dati. Grafana è un tool open source per realizzare dashboard generiche, attraverso cui è possibile creare grafici basati su dati memorizzati in InfluxDB. Negli esempi di app EF-Pi presenti nel repository https://github.com/flexiblepower/fpai-apps , sono implementati altri metodi che scrivono su un server di ricerca open source, chiamato ElasticSearch [9]. Per la visualizzazione dei dati registrati su questa piattaforma, è possibile utilizzare una dashboard open source chiamata Kibana [10].

Per ogni componente è possibile definire il relativo widget per l'interfaccia EF-Pi. I widget possono essere di vario tipo: ci sono i piccoli widget che compaiono sulla dashboard, oppure i widget a pagina intera. Tecnicamente, i widget sono degli iframes, pertanto ogni widget è in pratica esso stesso un piccolo sito web, all'interno del quale è possibile utilizzare le varie tecnologie come http, css, javascript, per renderlo interattivo. Sebbene EF-Pi metta a disposizione una serie di classi che aiutano a definire il proprio widget, questi sono interamente definiti dai componenti stessi.

3.4 Energy Flexiblity Interface

Il componente EFI è un protocollo di comunicazione che abilita gli utenti finali al controllo di svariati dispositivi intelligenti, quali ad esempio lavatrici, lavastoviglie, unità di condizionamento, pannelli solari, stazioni per la ricarica di veicoli elettrici, etc…rendendo queste risorse in grado di valorizzare la loro capacità di offrire flessibilità nel consumo o nella produzione dell'energia.

Questa interfaccia è attualmente in corso di standardizzazione a livello europeo, attraverso la norma tecnica EN50491-12 [11], in cui viene denominata come S2. L'idea alla base di questo approccio è quella di ottenere come risultato l'80% della flessibilità energetica con un 20% della complessità. In altre parole, l'intenzione non è quella di rendere l'EFI perfetta, bensì ottenere una modalità realmente pratica ed efficiente di collegare diversi apparati ad un sistema per la gestione dell'energia (EMS). L'interfaccia in questione è stata sviluppata come software plug and play, e questo la rende ovviamente uno strumento molto utile, soprattutto nei casi in cui ci si trova a dover gestire la configurazione di sistemi con un elevato numero di apparati, in quanto non occorre conoscere alcuna informazione specifica se non quelle che vengono gestite attraverso l'interfaccia EFI. L'integrità dei dispositivi è mantenuta, ovvero l'interfaccia non forza alcun controllo sul dispositivo. A quest'ultimo è lasciata la facoltà di ignorare le istruzioni che riceve dal sistema di gestione dell'energia, il quale a sua volta naturalmente, verrà informato dell'impossibilità di applicare quei comandi. L'unico obiettivo di EFI è la flessibilità energetica, pertanto attraverso questa interfaccia non è possibile gestire aspetti di altra natura quali ad esempio le sessioni di ricarica o i l'impostazione dei termostati intelligenti. L'interfaccia S2 è pensata per essere sviluppata come API Rest, ma tuttavia EFI al momento non implementa questo tipo di servizio. EFI è stata validata ed ulteriormente migliorata nel corso della sua applicazione in molti progetti pilota, pertanto risulta essere un componente sufficientemente stabile per poter essere utilizzata in ambiti applicativi.

L'architettura di S2 prevede al livello inferiore una *smart appliance*, che implementa al suo interno uno tra i tanti protocolli di comunicazione presenti sul mercato, come ad esempio Konnex, Zigbee, EEEBUs, Modbus, etc…che consentono di controllare il dispositivo. Al livello superiore si trova un modulo software chiamato *Resource Manager* o gestore delle risorse, che svolge in pratica il ruolo di "interprete", in quanto traduce le informazioni che scambia con il dispositivo intelligente in informazioni di flessibilità. Per poter fare ciò, in genere il gestore delle risorse ha bisogno di un modello del dispositivo; per esempio una lampada elettrica da 45W, è un oggetto che può assumere due stati possibili, ovvero accesso e spento, e che può comunicare il suo stato attraverso uno dei tanti protocolli di comunicazione qui elencati; quello che tuttavia interessa all'energy manager è il livello di potenza che questa lampada sta assorbendo, ovvero 0W o 45W. E' pertanto compito del gestore di risorse tradurre il concetto di "acceso" o "spento" in "sta consumando 0W", o "sta consumando 45W". Al livello superiore del gestore di risorse si trova il CEM, ossia il sistema di gestione dell'energia, un modulo software che può essere mandato in esecuzione su un dispositivo fisico installato presso un'abitazione o un certo edificio. In genere, un CEM dialoga con un certo numero di gestori di risorse, ognuno corrispondente ad un particolare dispositivo connesso attraverso l'interfaccia S2, ovvero l'EFI. L'architettura standard prevede la presenza di un CEM per ogni connessione di rete, dal momento che il CEM rappresenta l'edificio nei confronti della rete elettrica. Al livello superiore infine, gli EMS comunicano con entità esterne, rappresentate da moduli software che implementano logiche di varia natura, come ad esempio sorgenti di profili di prezzo, altri agenti PowerMatcher, etc…

L'operazione che in sintesi realizza l'interfaccia S2 è quella di mappare i vari comportamenti fondamentali di flessibilità descritti in precedenza, in quattro modalità operative, chiamate anche spazi di controllo. Di conseguenza EFI non è un protocollo, bensì quattro protocolli che implementano quattro distinte modalità di controllo.

La prima versione di EFI, è stata implementata in linguaggio JAVA, e risulta essere una parte di EF-Pi, il componente descritto nella sezione 3.3. La seconda versione è stata implementata in forma di protocollo basato su linguaggio XML, per superare in questo modo la dipendenza da uno specifico linguaggio di programmazione. Il protocollo di trasporto in genere utilizzato per lo scambio di messaggi EFI sono le *websocket*. L'ultima versione è infine quella relativa allo standard ufficiale a livello europeo, ridenominata in S2, migliorata in alcune parti, grazie anche al supporto di svariati partner industriali, ed implementata attraverso un *proof-of-concept* basato su *REST API Modeling Language*, una tecnologia molto simile ad *Open API* [12]. Sebbene molti nomi e diversi aspetti tecnici siano cambiati in queste versioni, i concetti alla base risultano essere comunque più o meno gli stessi. Pertanto, il passaggio dalla prima all'ultima versione non dovrebbe comportare nessun cambiamento sostanziale per chi attualmente la sta utilizzando, dal momento che gli algoritmi di fatto restano invariati, trattandosi di modifiche che afferiscono ad una differente rappresentazione dei concetti ed una differente codifica dei dati.

I nomi associati alle quattro categorie introdotte in precedenza cambiano da una versione all'altra dell'interfaccia, ma il loro sostanziale significato resta lo stesso. In questo contesto utilizzeremo gli stessi nomi che si trovano nell'implementazione JAVA attualmente disponibile. Nella categoria degli *Uncontrollable* sono ricondotti tutti gli apparati non direttamente controllabili, ma tuttavia in grado di fornire informazioni di interesse per la loro misurazione e predizione, come ad esempio gli impianti fotovoltaici, che in alcuni casi danno la possibilità di essere limitati in alcuni parametri. La seconda categoria fa riferimento ai cosiddetti *Time-Shiftable*, ovvero gli apparati che implementano dei programmi il cui avvio è comunque flessibile. Nella categoria dei *Buffer/Storage* ricadono quegli apparati in grado di bufferizzare o immagazzinare energia, come ad esempio sistemi di riscaldamento o raffreddamento, veicoli elettrici, batterie. Nella categoria dei cosiddetti *Unconstrained* ricadono invece tutti quegli apparati che presentano un'elevata flessibilità e consentono di essere modulati, come ad esempio gruppi elettrogeni o resistenze.

I vari concetti implementati all'interno di EFI sono riportati nello schema illustrato in Figura 7. Il blocco di colore verde, relativo alla flessibilità energetica, è il nucleo del componente. Attorno ad esso, si trovano quindi in colore arancione: le *Smart Grid Applications*, che identificano le varie *Customer Energy Resources* (CER), gli *Appliance Drivers*, che identificano i *Resource Managers*, i *Protocols*, che identificano i vari protocolli di comunicazione a basso livello, come ad esempio Modbus o Konnex. Questo stack di moduli software è in grado di comunicare con l'esterno, al livello inferiore con le varie *Appliances*, ed al livello superiore con i vari punti di controllo centrale delle smart grid. Per tradurre la flessibilità a livello superiore

si utilizza come visto il concetto di *Control Space*, mentre per tradurre la flessibilità a livello inferiore si utilizza il concetto di *Allocation*, ovvero il tipo di istruzione che si vuole venga eseguita dagli apparati.

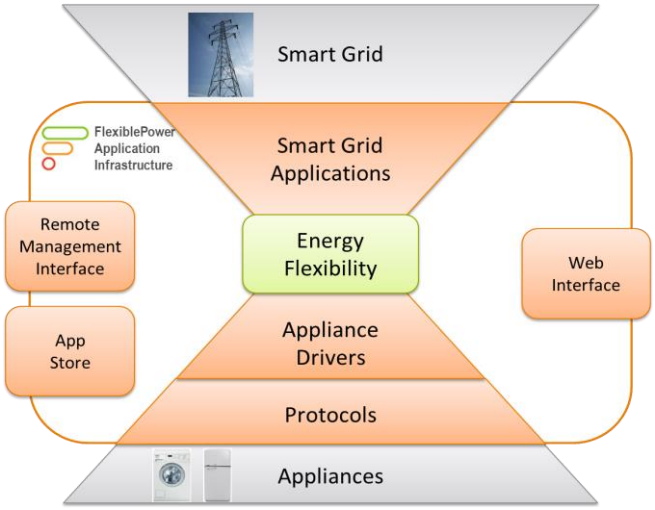

Figura 7: Architettura EFI (Fonte: [13])

In Figura 8 è rappresentato un tipico scambio di messaggi secondo lo standard EFI tra due elementi, il *Resource Manager* e l'*Energy Application*. Nell'implementazione utilizzata nell'ambito di questo progetto, i messaggi sono in pratica delle normali classi Java. Il processo attraverso cui avviene lo scambio di informazioni tra due elementi è più o meno sempre lo stesso, e comincia con l'invio di due messaggi: un *ControlSpaceRegistration* ed un *ControlSpaceUpdate*. Il primo contiene delle informazioni statiche relative al dispositivo che non verranno modificate nel corso della comunicazione. Il secondo contiene informazioni relative allo stato del dispositivo, come ad esempio la temperatura all'interno dell'edificio. L'*Energy Application* risponde in genere con un *Allocation*, ovvero un comando da far eseguire al dispositivo, come ad esempio "attiva" o "spegni il riscaldamento". In risposta, il *Resource Manager* può inviare un *AllocationUpdate*, in cui informa l'*Energy Application* dell'esito con cui si è concluso il comando richiesto, ovvero ad esempio "mi hai chiesto di accendere ed ho acceso il riscaldamento" oppure "mi hai chiesto di spegnere ed ho spento il riscaldamento". L'*Energy Application* può in genere inviare diverse richieste di *Allocation*, compresa quella di revoca del comando inviato in precedenza, attraverso una *AllocationRevoke*. Allo stesso modo, il *Resource Manager* può informare l'*Energy Application* di non essere più flessibile, attraverso l'invio di un *ControlSpaceRevoke*, come ad esempio potrebbe accadere nel caso in cui l'utente apra la porta di una lavatrice, che di conseguenza assume uno stato che la rende non più disponibile. La tipologia di questi messaggi è differente per ognuna delle quattro modalità di controllo analizzate in precedenza.

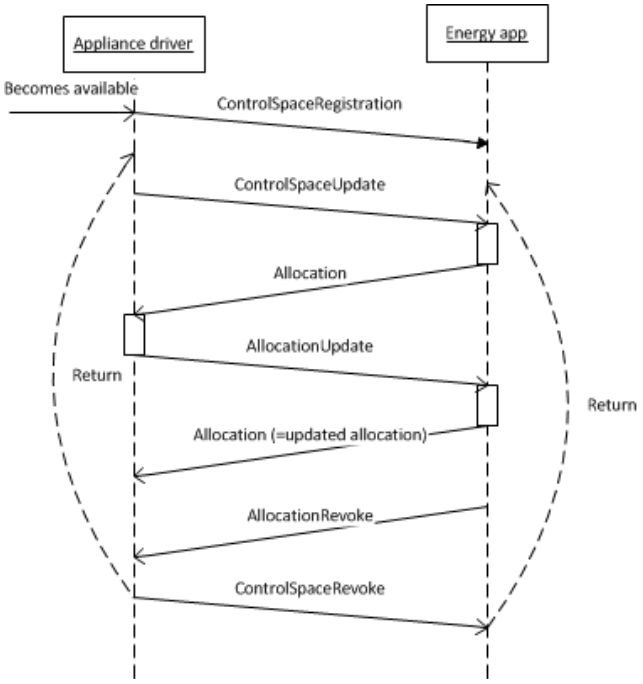

Figura 8

: Scambio di messaggi in EFI (Fonte: [6])

Bibliografia

[1] «vSphere,» [Online]. Available: https://www.vmware.com/products/vsphere-hypervisor.html.

[2] «PFSense,» [Online]. Available: https://www.pfsense.org/.

[3] «FlexiblePower,» [Online]. Available: http://flexiblepower.github.io/.

[4] «OSGi,» [Online]. Available: https://en.wikipedia.org/wiki/OSGi.

[5] K. Kok, «The PowerMatcher: Smart Coordination for the Smart Electricity Grid,» [Online]. Available: https://research.vu.nl/en/publications/the-powermatcher-smart-coordination-for-the-smart-electricity-gri.

[6] «EF-Pi,» [Online]. Available: https://fan-ci.sensorlab.tno.nl/builds/fpai-documentation/development/html/.

[7] «InfluxDB,» [Online]. Available: https://www.influxdata.com/.

[8] «Grafana,» [Online]. Available: https://grafana.com/.

[9] «Elastic Search,» [Online]. Available: https://github.com/elastic/elasticsearch.

[10] «Kibana,» [Online]. Available: https://github.com/elastic/kibana.

[11] «EN 50491,» [Online]. Available: https://ec.europa.eu/eip/ageing/standards/home/domotics-and-home-automation/en-50491_en.

[12] «OpenAPI,» [Online]. Available: https://www.openapis.org/.

[13] S. DAGIOGLOU, «An implementation of the planning phase of Triana using the flexible power application infrastructure,» 2014. [Online]. Available: <https://www.semanticscholar.org/paper/An-implementation-of-the-planning-phase-of-Triana-Dagioglou/2664d22b2fdc57e0ed6dcb87e322aa15766088bc.