Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Definizione dell'architettura software di un Sistema Cyber-Physical

Loris Belcastro, Andrea Giordano,
Giandomenico Spezzano,
Andrea Vinci

RT-ICAR-CS-13-05 **Novembre 2013**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.icar.cnr.it*

Sommario

I sistemi cyber-physical sono sistemi intelligenti che integrano componenti computazionali (hardware e software), componenti di comunicazione (rete) e componenti fisiche e meccaniche, in grado di “sentire” l'ambiente fisico ed interagire con esso, con l'obiettivo di monitorare e controllare un certo ambiente fisico o supportare le attività umane.

In questo documento è presentata una architettura distribuita per sistemi cyber-physical, che utilizza dei nodi di calcolo co-locati nello stesso ambiente ove sono presenti i dispositivi fisici. L’architettura nasconde l’eterogeneità dei dispositivi fisici attraverso l’astrazione dei Virtual Object, e prevede l’utilizzo di un middleware distribuito ad agenti sui nodi di calcolo. L’obiettivo è quello di avvicinare il più possibile la computazione ai dispositivi, riducendo le latenze di rete, e di permettere il pieno utilizzo di algoritmi decentralizzati e tecniche di swarm intelligence per il controllo degli ambienti fisici e la coordinazione dei dispositivi.

Indice

Elenco delle figure

Elenco dei codici

1 Introduzione

Le principali infrastrutture fisiche, quali, ad esempio, quelle per la gestione dell'acqua, dell'energia elettrica, dei trasporti, sono sistemi complessi a grande scala che devono essere altamente affidabili, per garantire costantemente una certa qualità del servizio, oltre che reattivi, per poter rispondere rapidamente a variazioni ambientali anche critiche, quali, ad esempio, una variazione repentina nella domanda di un servizio, un evento climatico inaspettato, o più semplicemente un guasto all'interno del sistema.

Instrumentando capillarmente i sistemi con un certo numero di dispositivi di sensing e di attuazione, e connettendo questi ultimi a servizi informatici che possano gestirli direttamente, è possibile dotare le infrastrutture di intelligenza. Attraverso i sensori, le infrastrutture sono capaci di osservare continuamente il proprio stato di funzionamento; servizi informatici possono utilizzare queste informazioni per elaborare strategie, politiche e direttive che ottimizzino l'uso delle risorse disponibili per soddisfare la domanda, o predire possibili guasti e prevenirli. Le direttive prodotte possono essere poi attuate sulla rete fisica attraverso l'utilizzo degli opportuni dispositivi di attuazione. Si costituisce, insomma, una sorta di sistema nervoso potenzialmente in grado di monitorare e gestire una infrastruttura su qualunque scala, a partire dai sistemi di refrigerazione di una catena di supermercati fino alle infrastrutture fisiche complesse prima citate.

Progettare sistemi di questo tipo vuol dire progettare sistemi intelligenti che integrino mondo fisico e mondo digitale, comprendendo componenti computazionali (hardware e software), componenti di comunicazione e componenti fisiche e meccaniche in grado di sentire ed agire sul mondo fisico, con l'obiettivo di monitorare e gestire un sistema fisico o supportare le attività umane [nist13].

Le ricerche per realizzare tale integrazione sono tuttora svolte parallelamente da due diverse comunità, con visioni e approcci differenti. La comunità che ruota intorno ai Cyber-Physical Systems (CPS) parte dal mondo fisico dei processi industriali e dei controlli automatici, mentre la comunità che porta avanti la visione dell'Internet-of-Things (IoT) parte dal mondo dell'elaborazione digitale ed informatica [jes13]. Mentre i CPS partono dal mondo fisico e muovono verso il mondo digitale, la visione IoT muove nel senso opposto. Entrambe le visioni hanno lo stesso obiettivo comune, ma i differenti approcci portano ognuno problematiche e soluzioni proprie.

1.1 Cyber-physical Systems

L'intuizione alla base dei Cyber-Physical Systems è quella di fornire meccanismi per la comunicazione ed il controllo attraverso l'utilizzo di due reti parallele: una rete fisica che interconnetta i componenti dell'infrastruttura, e una rete cyber, composta dai controllori intelligenti e dalla loro orchestrazione. Queste due reti interagiscono fra loro per monitorare e controllare i processi fisici.

I CPS [lee06][hua08][kou09][sci11][san12] rappresentano una struttura capace di amalgamare computazione e proprietà fisiche e sono diventati fondamentali per lo sviluppo delle applicazioni in molti domini. Nel mondo convergente cyber-physical, un'ampia varietà di smart devices, come RFID, sensori e attuatori, smartphone dotati di sensori, o tecnologie di sensing di prossimità, sono capaci di collezionare informazioni relative al comportamento dell'utente, i loro requisiti e la loro dinamica. Questo può contribuire a definire un ecosistema in cui i componenti individuali (cioè cittadini, device, servizi software, sensori e attuatori, smart object) sono in grado di collaborare per la realizzazione di avanzati servizi urbani. Tali servizi possono contribuire a realizzare la visione della smart city lungo parecchie dimensioni: dai sistemi di trasporto intelligenti, alla sostenibilità ambientale e alla governance partecipativa. Tali servizi avranno impatto sul nostro modo di vivere in ambienti urbani. La gestione di una tale infrastruttura complessa necessita lo sviluppo di politiche efficaci e scalabili per trattare la cooperazione fra questi dispositivi e adattare il loro comportamento al rapido cambiamento fisico (e.s. locazione e altre caratteristiche ambientali) e sociale (attività correnti e mutua interazione) dei contesti da cui i servizi digitali sono invocati.

Oltre l'idea di base di amalgamare il mondo fisico e quello virtuale (cyber) vi è quella di raccogliere tutte le informazioni utili circa gli oggetti del mondo fisico e usarle in varie applicazioni durante l'intero ciclo di vita degli oggetti. Collezionare informazioni e renderle disponibili, per esempio, circa l'origine degli oggetti e dei beni, la locazione, i movimenti, le proprietà fisiche, *usage history* e contesto, può aiutare a migliorare le applicazioni esistenti o a crearne di nuove.

Smart Grid La smart grid, ad esempio, può essere considerata come un complesso ecosistema di entità eterogenee cooperanti che interagiscono per offrire le funzionalità desiderate. Incorporando sensori e attuatori, la smart grid è in grado di monitorare e controllare l'infrastruttura ma necessita di strumenti avanzati di analytics che operano sopra milioni di dati streaming per prendere decisioni in maniera affidabile ed efficiente. Per realizzare la smart grid è necessario realizzare un sistema cyber-physical dove l'infrastruttura fisica e la cyber-infrastruttura

computazione devono cooperare per assicurare un funzionamento affidabile e ottimizzato.

La complessità è determinata dalla presenza di fonti rinnovabili come il vento e il solare che sono intrinsecamente inaffidabili e causano una erogazione incerta di elettricità. D'altra parte la presenza di elettrodomestici intelligenti, l'adozione di veicoli elettrici rende il profilo di carico del cliente variabile. Questi elementi causano instabilità nella rete che dovrebbero essere immediatamente rettificate. In assenza di un supporto computazionale e di strumenti di analytics per automatizzare le decisioni è molto difficile che gli operatori umani possano intervenire per definire punti di controllo capaci di gestire il dinamismo generato dall'utilizzo della rete elettrica.

Per rendere smart la rete sono richieste avanzate tecniche informatiche e infrastrutture cyber. Possono essere utilizzati eventi streaming da milioni di smart meter, campionati ogni 15 minuti, che necessitano di essere collezionati e correlati per fornire profili storici degli utenti. Tecniche di data mining e pattern matching sono necessarie per rilevare in tempo reale situazioni critiche e per attuare immediatamente le correzioni per garantire la stabilità della griglia. Analytics e modelli computazionali possono essere di ausilio per predire la potenza richiesta in un dato tempo.

Water Grid Anche le reti di distribuzione dell'acqua sono un dominio emergente dei CPS. Componenti fisici come valvole, condotte e bacini sono accoppiati con hardware e software che supportano la distribuzione intelligente dell'acqua. L'obiettivo di una rete di distribuzione idrica urbana è quello di fornire un sorgente affidabile di acqua potabile agli utenti. Rilevate attraverso gli opportuni sensori, le informazioni sui pattern di richiesta, sulla quantità (flusso e pressione) e sulla qualità (contaminanti e minerali) dell'acqua, possono essere elaborate dagli algoritmi di un sistema CPS per controllare direttamente le valvole e le pompe idrauliche della rete al fine di garantire un servizio efficiente, oppure per alimentare un sistema di supporto alle decisioni utilizzabile dall'ente che gestisce e mantiene la rete idrica.

1.2 Internet of Things

La convergenza fra ICT ed elettronica di consumo porta sempre più alla necessità di interpretare come "device in rete" oggetti molto diversi e non connessi

fra loro. Quello dell'internet delle cose (IOT) è destinato a divenire un settore fondamentale nella diffusione delle nuove tecnologie sia all'interno delle aziende sia nella vita quotidiana.

L'internet delle Cose propone un'evoluzione dell'attuale paradigma di comunicazione della rete Internet ancora dominato da uno scambio di informazioni fra strumenti molto simili fra loro (PC, Tablet, Smartphone) e che vede, quasi sempre, il coinvolgimento dell'utente umano come destinatario o sorgente dell'informazione. Nel nuovo paradigma gli oggetti acquisiscono la capacità di interagire tra loro, anche senza l'intervento dell'uomo e, necessariamente, aumentano le loro capacità di autoriconfigurarsi e auto governarsi, guidati da informazioni prelevate autonomamente dal contesto nel quale operano ("Context Awareness").

Al fianco delle classiche infrastrutture urbane (trasporti, illuminazione pubblica, approvvigionamento e distribuzione dell'energia, linee telefoniche cablate, etc.) si sta costruendo uno strato di oggetti e servizi interconnessi che abilitano la trasformazione delle città in ecosistemi intelligenti.

Grazie a questo nuovo "Digital Layer", che andrà a dotare oggetti fino ad oggi isolati delle capacità di dialogare in rete, e grazie alla sua integrazione con altri strati infrastrutturali, si stanno creando i presupposti per la realizzazione del concetto di Smart City di cui si parla ormai da tempo.

Si assisterà ad un sostanziale miglioramento della qualità della vita, soprattutto negli spazi urbani dove verranno allestiti sistemi in comunicazione fra loro. Questi potranno essere impiegati a supporto del traffico, o a sostegno del risparmio energetico, o per la gestione e la sicurezza di luoghi e persone, o per l'assistenza remota dei pazienti, o per la creazione di nuovi servizi per la città e il turismo.

1.3 Rainbow

A partire dal panorama fin qui descritto, proponiamo una piattaforma software per infrastrutture energetiche basata su un sistema cyber-physical con oggetti cooperanti assistiti da tecnologie Cloud. La piattaforma vuole essere adattativa, interoperabile, scalabile, event-driven e capace di operare in contesti multi-dominio.

La piattaforma, denominata **Rainbow**, offre un middleware adattativo per acquisire dati dinamici da sensori e smart object ed effettuare operazioni di controllo, un repository per condividere dati e risultati; modelli scalabili di machine-learning

addestrati sopra dataset massivi contenenti dati storici per un'agile previsione della domanda, strumenti di analytics, un web portal e mobile app per favorire l'interazione delle persone con il CPS.

Rainbow è una piattaforma innovativa di servizi cloud, device e middleware per gestire infrastrutture energetiche e renderle intelligenti.

Rainbow abilita un ecosistema di comunicazione fra persone, oggetti e servizi. Attraverso moduli componibili e configurabili, fornisce gli elementi necessari per la costruzione di soluzioni verticali e orizzontali, scalabili e flessibili che si basano sull'interazione e cooperazione di oggetti connessi fra loro.

2 Architettura di Rainbow

Le caratteristiche primarie di **Rainbow** sono pervasività, trasparenza, portabilità, flessibilità, sensibilità al contesto, auto-adattabilità ed auto-configurabilità.

Rainbow intende soddisfare i seguenti requisiti:

- Interoperabilità, attraverso l'utilizzo di protocolli standard per lo scambio dati con l'esterno della piattaforma.
- Facilità di configurazione, manutenzione e riconfigurazione, con un ridotto livello di interazione per abilitare nuovi servizi.
- Rapidità di reazione ad eventi critici o inaspettati attraverso la distribuzione dell'intelligenza su nodi di elaborazione vicini ai dispositivi di attuazione e di sensing.
- Monitoraggio di grandezze per applicazioni specifiche (ad esempio parametri biomedici, ambientali, di funzionamento apparati, etc.).
- Attuazione comandi a fronte di eventi o azioni di controllo (ad esempio comandi in ambienti domotici, industriali, etc.).
- Capacità di essere utilizzata in diversi domini applicativi.

Attualmente, l'approccio più utilizzato nella progettazione di systemi cyberphysical prevede una strutturazione su due livelli: uno legato più intimamente all'ambiente fisico, composto da sensori ed attuatori in esso immersi, ed uno computazionale, generalmente posto su cloud. Attraverso la rete, i dati (le misure)

fluiscono dai sensori al cloud. Nel cloud vengono analizzati ed elaborati, quindi inviati gli opportuni comandi agli attuatori sul livello più basso. Questo approccio, in contesti di grande scala, presenta alcuni limiti individuati nella occupazione di banda di trasmissione (necessaria per far fluire tutti i dati nel livello di computazione), nei tempi di risposta relativamente lenti, dovuti al delegazione delle elaborazioni e decisioni su cloud, e nella scalabilità degli algoritmi utilizzati per l'analisi dei dati.

Per affrontare e risolvere queste problematiche e per rispondere ad i requisiti descritti in precedenza, ci si è mossi fondamentalmente in tre direzioni:

- spostando la computazione più vicina possibile al mondo fisico (sensori ed attuatori);
- introducendo un livello di intelligenza distribuita tra mondo fisico e cloud, in grado di eseguire compiti complessi e coordinare i vari dispositivi;
- passando da un modello cloud-based ad uno cloud-assisted: il sistema utilizza le capacità del cloud soltanto per analisi e supporto alle decisioni.

Spostando la computazione il più vicina possibile alle risorse fisiche, si vuole ridurre la portata dei flussi di dati sulla rete, eseguendo un primo filtraggio delle misure di interesse, e diminuire i tempi di risposta per le attività che non necessitano di coordinazione tra più elementi del sistema. Muovendo in questa direzione ci siamo scontrati con l'attuale eterogeneità dei dispositivi presenti. Non tutti i dispositivi, inoltre, sono in grado di eseguire computazione. Per risolvere questa problematiche, abbiamo considerato un layer di virtualizzazione: un virtual object è un aggregato di sensori ed attuatori con minima capacità di calcolo, che fornisce all'utilizzatore della piattaforma un insieme di funzionalità e servizi omogenei. In mancanza di un componente fisico che possa effettuare computazioni semplici, l'elaborazione è trasparentemente effettuata sui nodi di calcolo sui quali risiede il livello di intelligenza distribuita.

Introducendo un livello di intelligenza distribuita tra il mondo fisico e quello su cloud, diminuiamo il carico e la latenza di rete, aumentando al contempo l'affidabilità del sistema. Per affrontare le problematiche di cooperazione e coordinazione degli oggetti, ci affidiamo ad un sistema distribuito ad agenti software. Un paradigma ad agenti software è particolarmente adatto alla implementazione di forme di intelligenza distribuita attraverso la progettazione di algoritmi p2p e di swarm intelligence, a conoscenza incompleta, orientati al controllo di un certo sistema fisico. Approcci ad agenti per la progettazione di CPS sono utilizzati, in modi diversi dal nostro, anche in altre soluzioni esistenti [tal08][lei13][lin10][for13].

Il cloud rimane comunque importante nella piattaforma pensata, seppur ridimensionato nelle sue funzioni alle attività di analisi predittive e supporto alle decisioni. In generale, quindi, un'applicazione complessa deve essere deployata in modo da collocare le attività prevalentemente "*data intensive*" sui nodi direttamente connessi alle sorgenti di informazioni d'interesse e le attività prevalentemente "*computational intensive*" sui nodi del cloud. Attività "data intensive" possono essere, ad esempio, il filtraggio dei dati di interesse, l'aggregazione, la compressione dei dati, etc. Attività "computational intensive" possono essere, ad esempio, algoritmi per il supporto alle decisioni, simulazioni, mining di dati storici, etc.

Sintetizzando, possiamo definire quindi la nostra piattaforma per cyber-physical systems come composta da tre livelli, mostrati in Figura 1:

- un primo livello fisico, costituito da aggregati di sensori e attuatori, con limitate capacità di calcolo;
- un secondo livello formato da nodi computazionali, che ospita principalmente il modulo per la gestione dei Virtual Object e quello per l'esecuzione degli agenti software, che gestisce in tempo reale il livello fisico;
- un terzo livello sul cloud, che possa assistere il secondo con analisi predittive, data mining ed elaborazioni non in tempo reale.

La presenza di sensori fisici suggerisce che l'infrastruttura utilizzi un paradigma basato su *eventi*. Gli eventi possono essere semplici (direttamente generati dai VO) o complessi (ovvero calcolati in base all'occorrenza di diversi eventi semplici).

In Figura 2 si mostra come da eventi semplici (E) generati dai VO vengono generati eventi complessi (Ec) tramite opportuno reasoning.

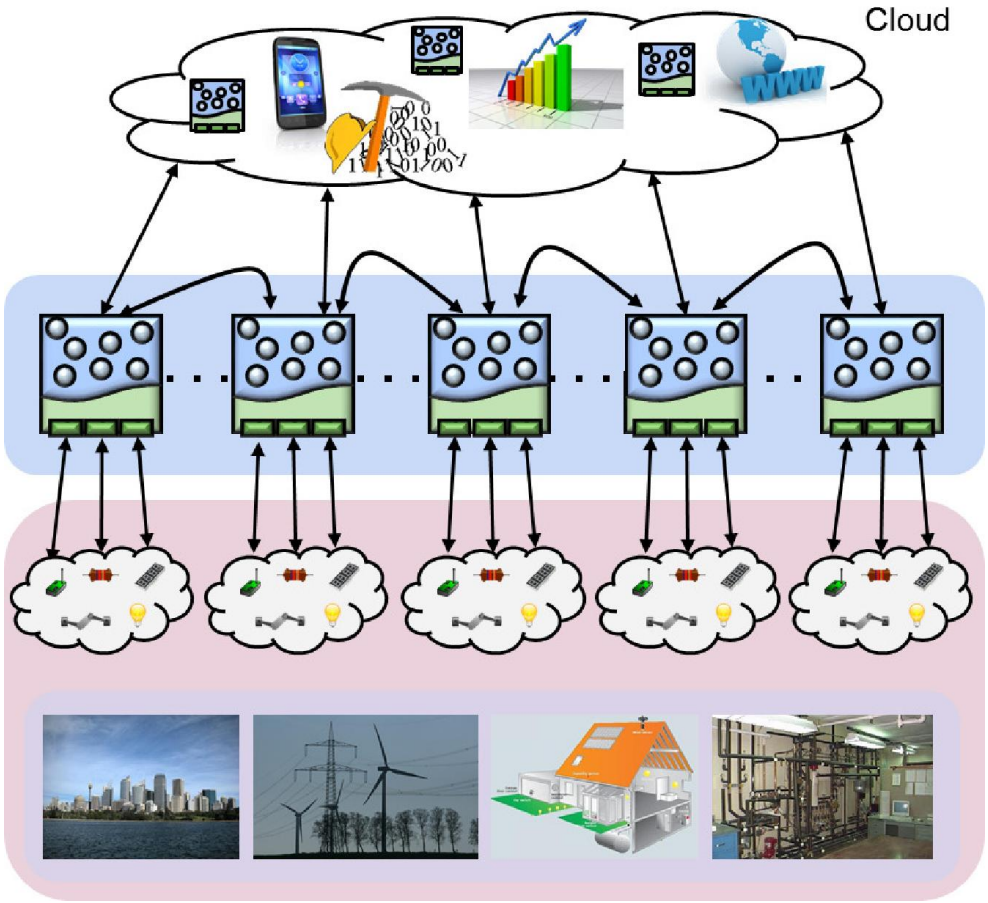

Figura 1: Architettura di Rainbow.

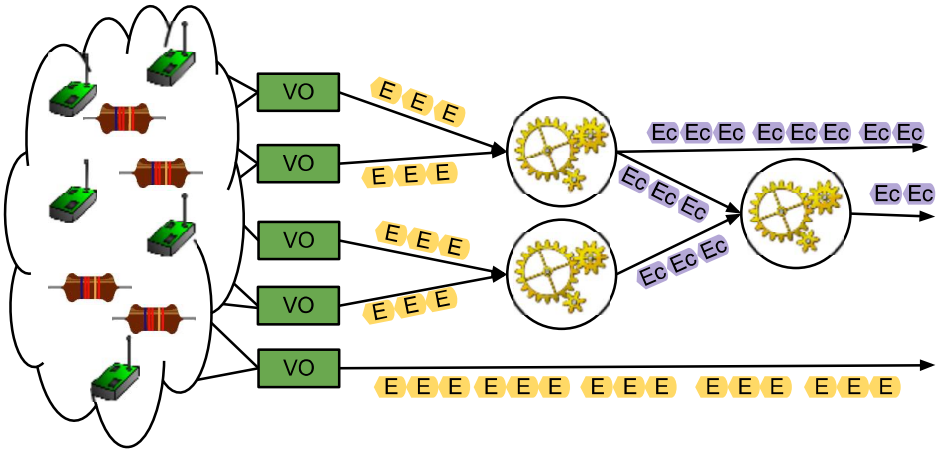

Figura 2: Processamento di eventi.

3 Infrastruttura software dei nodi di calcolo

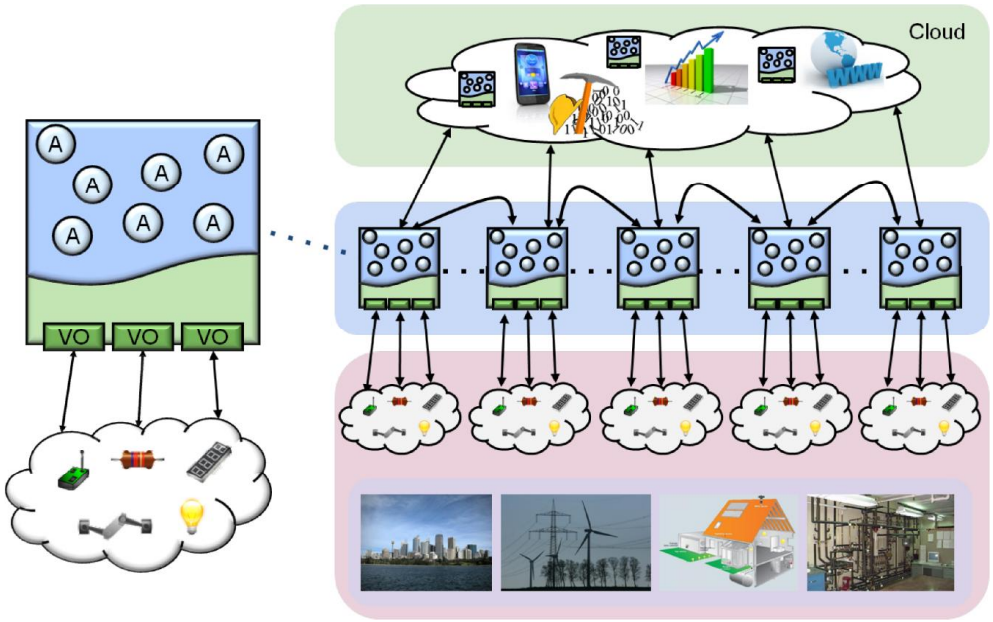

Figura 3: Nodo di calcolo in Rainbow.

In questa sezione ci occuperemo di descrivere la piattaforma software resa disponibile per l'esecuzione di applicazioni cyber-physical ad agenti. Ogni nodo computazionale, direttamente connesso con i dispositivi fisici (sensori e attuatori), conterrà l'ambiente Rainbow.

Come descritto in precedenza, le risorse fisiche sono wrappate dai Virtual Obejct. I VO utilizzano le risorse fisiche per esporre servizi ai livelli superiori tramite una interfaccia ben stabilita. Se consideriamo di avere, ad esempio, diversi sensori di temperatura in una stanza, un VO potrebbe, utilizzando i sensori, esporre un servizio che indichi la temperatura in un singolo punto della stanza.

I VO sono gestiti da un contenitore, detto *gateway*. Il gateway è un processo che gira sul nodo al quale sono connesse le risorse fisiche e conosce i servizi esposti dai VO contenuti.

Il gateway contiene al suo interno anche la componente che gestisce il *publish/subscribe* di eventi, esponendo, quindi, delle funzioni per la pubblicazione e sottoscrizione di *regole* che coinvolgono, in generale, più servizi e funzioni ap-

partenenti a diversi virtual object.

Su ogni nodo di calcolo, oltre al gateway, è presente l'*agent server*. Esso permette l'esecuzione di agenti caricati anche da remoto e, nel contempo, espone funzioni che consentono le comunicazioni tra agenti, sia localmente che remotamente. Gli agenti incapsulano, in modo distribuito e ubiquo, la logica applicativa e possono accedere alle risorse fisiche tramite l'interfaccia esposta dal gateway.

In tal senso, si può definire uno smart object come un insieme di risorse fisiche, oggetti virtuali e agenti cooperanti tra loro.

Un nodo non contenente oggetti virtuali è un nodo puramente computazionale. Tali nodi, non avendo necessità di essere localizzati vicino alle risorse fisiche, possono risiedere ovunque. Ciò offre l'opportunità di sfruttare servizi di *cloud computing* per avere la potenza di calcolo necessaria.

3.1 Virtual Object vs risorse "fisiche"

Il Virtual Object, come già detto, può incapsulare un semplice sensore, che fornisce una grandezza fisica, oppure una struttura molto più complessa (es. nel caso di smart room, smart building etc.). In generale, un Virtual Object può fornire diverse informazioni, siano esse *puntuali* (es. la temperatura in un dato punto nello spazio) che *aggregate* (es.l'umidità media nell'arco delle ultime 24 ore). Le informazioni del Virtual Object possono scaturire da semplici misurazioni fisiche (es. il valore di un certo sensore) così come possono essere frutto di elaborazioni complesse o che coinvolgano più entità insieme (es. la temperatura in un dato punto geografico stimata interpolando le misurazioni fornite da sensori disseminati nel territorio).

Il Virtual Object, inoltre, può avere funzionalità di *attuazione*, modificando lo *smart environment* che lo circonda in base a stimoli esterni o propri reasoning.

I Virtual Object possono anche essere organizzati in modo gerarchico: una *applicazione* che utilizza diversi Virtual Object (anche non relativi allo stesso nodo di calcolo) può implementare a sua volta la Virtual Object Interface, divenendo anch'esso un Virtual Object. Ciò può essere utile per fornire risorse aggregate e/o frutto di reasoning.

Tale generalità di comportamenti deve riflettersi nel modo in cui si interagisce con i Virtual Object. Per tali motivazioni, si è pensato di considerare il Virtual

Object come un oggetto complesso che può fornire, o attuare su, diverse risorse fisiche elementari.

In tale contesto, si è previsto di utilizzare la metafora dei *servizi*, qui intesi più propriamente come *funzionalità*. Ogni Virtual Object espone dei servizi (es. una Virtual Room può offrire la temperatura di una stanza, la sua umidità, il numero medio di persone che vi entra ogni giorno etc.).

Per generalità, si pensa che ogni servizio/funzionalità, sia esso di *sensing* o *acting*, possa essere configurabile, nel senso che è possibile passare dei parametri per specificare dinamicamente il suo comportamento.

Da queste considerazioni si prevede che la risorsa elementare utilizzabile sia identificata dalla seguente tripletta:

$$[VOId, VOServiceId, VOServiceParams]$$

dove *VOId* identifica il Virtual Object, *VOServiceId* identifica il servizio/funzionalità e *VOServiceParams* è, in generale, una lista di parametri per la configurazione del servizio scelto.

Esempio. Si supponga di disporre di un Virtual Room che incapsula delle grandezze fisiche all'interno di una stanza. Tale Virtual Object è realizzato con diversi sensori disseminati nella stanza. Se dall'esterno vogliamo leggere la temperatura in un preciso punto della stanza, la tripletta potrebbe essere del tipo: $[VirtualRoom, temperature, [x, y, z]]$.

3.2 Publish/subscribe di eventi

La componente di *publish/subscribe* prevede la pubblicazione e la sottoscrizione di eventi tramite delle regole che ne definiscono l'occorrenza. Ogni regola può interessare più "servizi" dello stesso Virtual Object o di differenti Virtual Object. Una regola è definita come una *proposizione logica* i cui *atomi* sono del tipo:

- $risorsa\ \text{“}fisica\text{”} < soglia$ (es. $temperatura < 300$),
- $risorsa\ \text{“}fisica\text{”} > soglia$,
- $risorsa\ \text{“}booleana\text{”}$ (es. $porta\_aperta$).

Un esempio di regola:

$$(temperatura < 100 \vee luminosità > 500) \wedge num\_persone > 3 \wedge porta\_aperta$$

La regola, comunque complessa, verrà parserizzata in modo da generare un albero binario costruito nel seguente modo: ad ogni nodo dell'albero è associata una proposizione logica; i figli f_L e f_R di un nodo n sono associati a proposizioni logiche $F_{Ln}()$ e $F_{Rn}()$, in modo tale che la proposizione $N()$, associata al nodo n, sia pari a $N() = F_{Ln}() \wedge F_{Rn}()$ oppure $N() = F_{Ln}() \vee F_{Rn}()$. La radice dell'albero è associata alla regola totale, mentre le foglie sono associate alle proposizioni atomiche passate ai Virtual Object. Essi sono incaricati di stabilire, istante per istante, se una data proposizione è soddisfatta o meno (Figura 4).

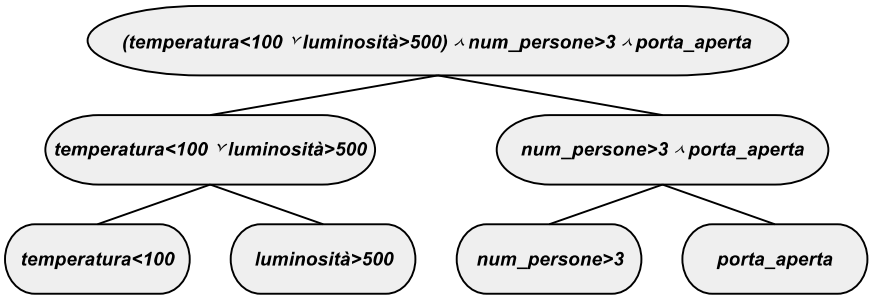

Figura 4: Esempio di risoluzione di una regola.

Le proposizioni dei nodi vengono valutate in base al valore delle proposizioni dei figli. Quando la proposizione associata alla radice è soddisfatta viene notificata ai sottoscrittori.

3.3 Interfaccia per i Virtual Object

Dalle considerazioni precedenti, l'interfaccia dei Virtual Object utile per interagire con l'architettura descritta è mostrato nel Codice 1, dove: *checkValue* è il metodo per leggere una data risorsa "fisica"; *streamValue* consente di ottenere il flusso dati di una certa risorsa; *acting* permette di eseguire una operazione di attuazione che produce una modifica nello smart environment; mentre i due metodi *setRule* servono ad impostare le regole atomiche (vedi sezione 3.2).

VOServiceId e *VOServiceParams*, dato un Virtual Object, identificano univocamente una risorsa "fisica", come spiegato precedentemente. *Operator* in *setRule*

Codice 1: Interfaccia per Virtual Object.

```
interface VirtualObjectInterface {
    VOResult checkValue(VOServiceId serviceId, VOServiceParams
        params);
    void streamValue(VOServiceId serviceId, VOServiceParams params,
        VONewValueListener listener);
    VOResult acting(VOServiceId serviceId, VOServiceParams params);
    void setRule(VOServiceId serviceId, VOServiceParams params,
        Operator operator, VOValue threshold, RuleMatchedListener
        listener);
    void setRule(VOServiceId serviceId, VOServiceParams params,
        RuleMatchedListener listener);
}
```

può valere banalmente "$>$" oppure "$<$" (maggiore o minore), mentre *threshold* è il valore di soglia.

L'ultimo parametro di *setRule* è un oggetto "*ascoltatore*" che viene notificato quando la regola è soddisfatta o meno, invocando uno dei metodi della sua interfaccia, descritta nel Codice 2.

Codice 2: Interfaccia RuleMatchedListener.

```
interface RuleMatchedListener {
    void ruleMatched();
    void ruleNotMatched();
}
```

Esempio Considerando la Smart Room usata precedentemente come esempio, se il predicato atomico deve essere ritenuto soddisfatto quando la temperatura per le coordinate [4,4,5] è minore di 27 gradi, allora il metodo *setRule* sarà invocato come nel Codice 3.

Oltre a implementare tale interfaccia, il Virtual Object dovrà sottoscriversi al gateway con una istruzione del tipo descritto nel Codice 4, fornendo così il suo identificativo univoco.

Codice 3: Esempio invocazione setRule.

```
//pseudo codice
SmartRoom.setRule(temperatura_Id, [4,4,5],Operator.minore, 27,
    OggettoCheVerraNotificato);
```

Codice 4: Registrazione di un virtual object ad un gateway.

```
//pseudo codice
Gateway.register(VirtualObjectId, this);
```

3.4 Interfaccia Gateway

Il gateway espone l'interfaccia descritta nel Codice 5.

Codice 5: Interfaccia del Gateway.

```
interface GatewayInterface {
    VirtualResource getResource(String name, VOId VoId, VOServiceId
        serviceId, VOServiceParams params);
    VOResult check(VirtualResource vr);
    VOResult check(VirtualResource vr, VOServiceParams params);
    void streamValue(VirtualResource vr, VONewValueListener listener
        );
    void streamValue(VirtualResource vr, VOServiceParams,
        VONewValueListener listener);
    VOResult acting(VirtualResource vr, VOServiceParams params);
    VOResult acting(VirtualResource vr);
    void setRule(Rule rule, IDRule idRule);
    void subscribe(IDRule idRule, VOEventListener listener);
}
```

Il metodo *getResource* restituisce un oggetto *VirtualResource* che incapsula una risorsa dei VO, cioè una tripletta $[VOId, VOServiceId, VOServiceParams]$. Tale oggetto identifica univocamente la risorsa negli altri metodi del gateway.

Dato un certo VirtualResource *vr*, il metodo *check* restituisce il valore della risorsa da esso identificata; il metodo *acting* innesca l'operazione di attuazione sulla risorsa; il metodo *streamValue* consente invece di ottenerne il flusso dati. Per questi metodi esiste la possibilità di passare opzionalmente dei parametri in

modo da configurare dinamicamente la risorsa alla quale ci si riferisce.

Il metodo *setRule* consente di pubblicare una regola complessa come descritto precedentemente (es: $(temperatura < 100 \vee luminosità > 500) \wedge num\_persone > 3 \wedge porta\_aperta$), e associare ad essa un identificativo che dovrà essere usato dai sottoscrittori. Il metodo *subscribe* consente di sottoscrivere una regola precedentemente pubblicata.

3.5 Agent Server

L'agent server espone metodi che consentono l'istanziazione (locale e remota) di agenti, e la comunicazione tra gli agenti stessi. L'interfaccia è descritta nel Codice 6.

Codice 6: Interfaccia dell'Agent Server.

```
interface CPAgentServer {
    String sendAgent(String agentClass, String name, HostPort addr);
    String sendAgent(String agentClass, String name, HostPort addr,
        HostPort classServer);
    String sendClass(String agentClass, HostPort addr , HostPort
        classServer);
    void send(CPMessage msg, HostPort addr);
}
```

Istanziare un agente su un Agent Server Per istanziare un agente su un agent server, sono disponibili i due metodi *sendAgent*. Entrambi richiedono una stringa *agentClass* contenente il nome della classe, una stringa *name* contenente il nome identificativo dell'agente e l'indirizzo del server sul quale lo si vuole istanziare.

Il nome di un agente lo identifica univocamente su quell'agent server.

Un esempio è mostrato nel Codice 7, nel quale un processo istanzia un oggetto della classe *smartRoom.AgentControlRoom*, di nome *pippo*, sull'agent server remoto 150.45.63.101:8081.

Codice 7: Esempio instanziazione di un agente.

```
CPAgentServer agentServer = ...;

String agentClass="smartRoom.AgentControlRoom";
String agentName="pippo";
HostPort destinationServer=new HostPort("150.45.63.101",8081);

agentServer.sendAgent(agentClass,agentName,destinationServer);
```

Istanziare un agente con migrazione del codice La precedente invocazione presuppone che la classe *smartRoom.AgentControlRoom* sia già nel classpath del server. Se ciò non fosse, la piattaforma rende disponibili due modi per caricare da remoto il codice di un agente.

Il primo consiste nell'utilizzo della variante del metodo *sendAgent* che richiede come parametro anche la posizione di un class server remoto. Un esempio è riportato nel Codice 8: in questo caso, l'agent server di destinazione richiederà il codice dell'agente *smartRoom.AgentControlRoom* al class server specificato.

Codice 8: Istanziazione di un agente con migrazione del codice I.

```
CPAgentServer agentServer = new CPAgentServerImpl();

String agentClass="smartRoom.AgentControlRoom";
String agentName="pippo";
HostPort destinationServer=new HostPort("150.45.63.101",8081);
HostPort classServer=new HostPort("150.45.63.102",8083);

agentServer.sendAgent(agentClass, agentName, destinationServer,
    classServer);
```

Il secondo modo è quello di precaricare la classe dell'agente tramite il metodo *sendClass*, rendendola disponibile per le successive istanziazioni. Un esempio nel Codice 9: mediante *sendClass*, si richiede all'agent server specificato di caricare la classe *agentClass* dal class server senza istanziare nessun oggetto. L'agente verrà istanziato in seguito attraverso l'invocazione del metodo *sendAgent*.

Il caricamento di classi da un class server remoto è una operazione, in generale, temporalmente dispendiosa. Tale operazione, durante l'esecuzione di una

Codice 9: Istanziazione di un agente con migrazione del codice II.

```
CPAgentServer agentServer = ...;

String agentClass="smartRoom.AgentControlRoom";
String agentName="pippo";
HostPort destinationServer=new HostPort("150.45.63.101",8081);
HostPort classServer=new HostPort("150.45.63.102",8083);

agentServer.sendClass(agentClass, destinationServer, classServer);
agentServer.sendAgent(agentClass,agentName,destinationServer);
```

applicazione, può prodursi in un "*freezing*" della stessa, con inevitabile ed imprevedibile calo prestazionale. L'utilizzo del secondo modo, invece, consente di precaricare tutte le classi utilizzate da una certa applicazione prima di eseguirla, eliminando la problematica appena descritta.

Inviare un messaggio ad un agente L'invio di un messaggio ad un agente si effettua tramite il metodo *send* dell'interfaccia nel Codice 6. Questo invia un messaggio all'agente identificato da *agentName* locato sull'agent server in esecuzione sul nodo identificato da *AgentServerAddr*.

Nell'esempio del Codice 10 si invia un messaggio all'agente *pippo* residente su un server remoto.

Codice 10: Esempio di invio di un messaggio ad un agente.

```
CPAgentServer localAgentServer =...;
String agentName="pippo";
HostPort remoteAgentServerAddr=new HostPort("150.45.63.81", 8083);
CPMessage msg=new CPMessageImpl();
localAgentServer.send(msg, agentName, remoteAgentServerAddr);
```

3.6 Message

Ogni messaggio deve implementare l'interfaccia *CPMessage* (Codice 11), alla base delle comunicazioni che avvengono tra gli agenti. L'informazione necessaria

è l'identificativo dell'agente destinatario del messaggio.

Codice 11: Interfaccia CPMessage.

```
public interface CPMessage{
    String getIdReceiver();
}
```

3.7 Agent

L'agente è responsabile della *smartness* dell'applicazione installata sulla piattaforma. Ogni agente è identificato da un id (nome) localmente univoco ed il suo comportamento è influenzato dalla ricezione di messaggi provenienti dai suoi pari (locali o remoti).

Codice 12: Interfaccia CPAgent.

```
public interface CPAgent {
    void receive(CPMessage msg);
}
```

Ogni agente deve implementare l'interfaccia mostrata nel Codice 12. In particolare, l'implementazione del metodo *receive* stabilisce il comportamento dinamico dell'agente. L'agent server è incaricato di consegnare i messaggi invocando tale metodo.

3.8 Deployment di un'applicazione e ruoli

La configurazione e l'avvio di un'applicazione ad Agenti che utilizzi la piattaforma, è demandata ad una componente, detta Deployer. Esso è responsabile dell'istanziazione degli agenti sui vari nodi e della configurazione delle associazioni tra gli agenti.
In generale, si prevede che un agente conosca l'identificativo ed il ruolo degli agenti con i quali dovrà comunicare. Questa conoscenza è data opportunamente dal Deployer attraverso l'invio di AcquaintanceMessage (descritti nel (Codice 13)).
Se un agente A associa un secondo agente B ad un certo ruolo, A interagirà con

Codice 13: Acquaintance Message.

```
public class AcquaintanceMessage extends CPMessageImpl {
    AgentLocation agente;
    String role;
}
```

B secondo le modalità che l'applicazione prevede per quel ruolo.

Due esempi di Deployment e utilizzo dei ruoli sono mostrato qui di seguito:
Esempio 1. Supponiamo di avere un'applicazione composta da un agente di tipo Master ed N agenti di tipo Slave, con l'agente Master che assegnare dei task agli Slave, e gli Slave che elaborano e restituiscono un risultato al Master.
Durante la fase di configurazione e avvio, il *Deployer* instanzia un agente Master su un certo nodo, e gli N agenti Slave su altri nodi. Dopo averli instanziati, invia all'agente Master un *AcquaintanceMessage* per ogni Slave. Attraverso questo messaggio, Master associa agli agenti il ruolo Slave. Contemporaneamente, il *Deployer* invia degli *AcquaintanceMessage* agli Slave, indicando qual è l'agente Master al quale devono riferire e dal quale possono accettare messaggi di tipo Task. Effettuate le interconnessioni, l'applicazione può essere avviata, mediante un messaggio di Start.
Esempio 2. Si vuole ottenere una configurazione gerarchica degli agenti, secondo una topologia ad albero. In questo caso, si considera una sola tipologia di agente, Nodo, e due ruoli, child e parent. In fase di configurazione, il Deployer instanzia un certo numero di agenti, quindi invia gli AcquaintanceMessage opportunamente. Ogni agente nodo verrà informato così di quali sono i suoi child e di quale nodo sia il suo parent.

Si noti che, se si dovesse avere la necessità di aggiungere dinamicamente ed a runtime agenti di tipo, ad esempio, Slave, basterà instanziare nuovi agenti di quel tipo e associarli al Master attraverso *AcquaintanceMessage*.
Oltre a questo, il meccanismo dei ruoli permette un'ampia flessibilità nel numero e nel tipo di ruoli, nonché nella cardinalità delle associazioni.

Con il framework è fornita una classe di utilità per gestire le *relazioni di conoscenza* ed i *ruoli* all'interno di un'applicazione ad agenti. La classe *Roles* (Codice 14) è utilizzabili a tale scopo. Questa consiste in una struttura dati in grado di gestire associazioni agente-ruolo, e metodi per la ricerca di agenti associati ad un certo ruolo o dei ruoli associati ad un certo agente. La classe fornisce anche un metodo per la gestione di un messaggio di tipo *AcquaitanceMessage*, descritta

Codice 14: Roles.

```
public interface RolesInterface extends Serializable{

    // Restituisce il set degli agenti conosciuti che hanno un certo
        ruolo.
    Set<AgentLocation> getAgents(String role);

    //Restituisce il set dei ruoli associati ad un agente conosciuto
        .
    Set<String> getRoles(AgentLocation agentLocation);

    //Aggiunge l'associazione di un agente ad un certo ruolo.
    boolean addAgentRole(AgentLocation agentLocation, String role);

    //Rimuove l'associazione di un certo agente ad un certo ruolo.
    boolean removeAgentRole(AgentLocation agentLocation, String role
        );

    //Verifica la presenza di una associazione agente-ruolo.
    boolean containsAgentRole(AgentLocation agentLocation, String
        role);

    //Rimuove dagli agenti conosciuti un certo agente.
    boolean removeAgentRole(AgentLocation agentLocation);

    //Tutti i ruoli conosciuti.
    public Set<String> getAllRoles();

    //Tutti gli agenti conosciuti.
    public Set<AgentLocation> getAllAcquaintancedAgents();

    //Gestisce appropriatamente un messaggio di conoscenza.
    void acquaintanceMsgReceived(AcquaintanceMessage msg);
}
```

nel (Codice 13).

4 Esempio di applicazione

In questo esempio vogliamo esaminare una possibile applicazione per il monitoraggio e il controllo di un intero piano di un edificio adibito ad uffici. Il piano è composto da un certo numero di stanze, all'interno delle quali sono dislocati sensori ed attuatori.

La topologia dell'ambiente preso in considerazione è mostrata in Figura 5. Ogni stanza contiene, in generale, una o più porte di ingresso/uscita, diverse scrivanie con le relative sedie e luci ad intensità variabile.

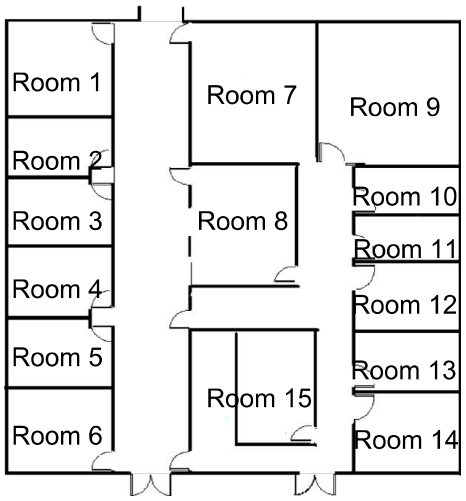

Figura 5: Topologia dell'esempio.

Ognuna di queste stanze è equipaggiata con diversi sensori e attuatori descritti di seguito.

Sensori:

- sensore che scatta quando la porta viene aperta o chiusa;
- sensore che identifica l'ingresso o l'uscita di una persona;

- sensori di prossimità che identificano la presenza di persone relativamente alle varie zone della stanza e anche vicino le scrivanie e le sedie;
- sensore di peso sulle sedie per identificare se la sedia è occupata o meno.

Attuatori:

- dispositivi per la variazione della luminosità delle luci relative alle varie zone della stanza;
- display collocato su ogni scrivania.

Utilizzando tali dispositivi si può, ad esempio, regolare l'illuminazione della stanza in base alla presenza delle persone, scrivere sui display informazioni d'interesse, etc.

4.1 Integrazione con il middleware

L'integrazione della parte fisica con il middleware avviene tramite la definizione dei Virtual Object che astraggono e incapsulano gruppi di sensori e attuatori. Per scelta progettuale i Virtual Object previsti si riferiscono ad entità umanamente riconoscibili: virtual desk, virtual chair, virtual door e virtual wall.

I servizi (funzionalità) esposte da tali VO sono descritti nelle Tabelle 1, 2, 3, 4. Si noti che Virtual Wall è parametrico: ognuna delle sue risorse si può riferire ai 4 quadranti della stanza in base al parametro zona: $0 \rightarrow$ alto-sinistra, $1 \rightarrow$ alto-destra, $2 \rightarrow$ basso-sinistra, $3 \rightarrow$ basso-destra.

Risorsa	Tipologia	Descrizione
lock	Sensing	Booleano, true se la porta è aperta
unlock	Sensing	Booleano, true se la porta è chiusa
ingresso	Sensing	Booleano, true se una persona entra dalla porta
uscita	Sensing	Booleano, true se una persona esce dalla porta

Tabella 1: Virtual Door.

Ogni VO risiede sul nodo di calcolo al quale sono connessi i sensori e gli attuatori che il VO incapsula. Più stanze possono, in generale, riferirsi ad uno stesso nodo di calcolo. Supponendo di disporre di tre nodi per il controllo dell'intero piano, una possibile assegnazione delle stanze ai nodi è mostrata nella Figura 6. Ogni colore diverso identifica un diverso nodo.

Risorsa	Tipologia	Descrizione
prossimita	Sensing	Rilevamento persone vicino la sedia
utilizzazione_sedia	Sensing	Booleano: true quando qualcuno è seduto

Tabella 2: Virtual Chair.

Risorsa	Tipologia	Descrizione
num_persone_vicine	Sensing	Numero di persone vicine alla zona
add_light	Acting	Aumenta la luce della zona
less_light	Acting	Diminuisce la luce della zona
light_off	Acting	Spegne la luce della zona

Tabella 3: Virtual Wall.

Risorsa	Tipologia	Descrizione
num_persone_vicine	Sensing	Numero di persone vicine alla scrivania
display	Acting	Mostra sul display la stringa passata come parametro

Tabella 4: Virtual Desk.

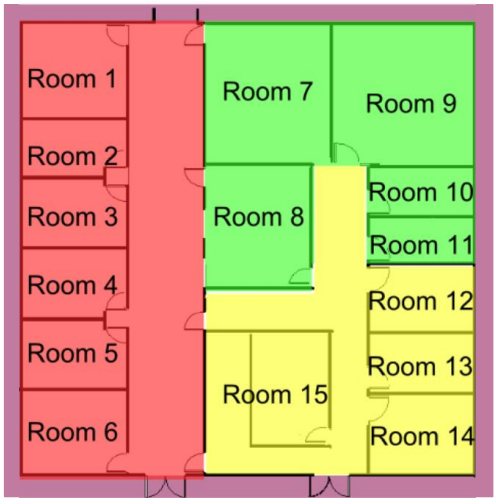

Figura 6: Assegnazione stanze ai nodi di calcolo.

4.2 Applicazione

Si vuole realizzare l'applicazione in modo da gestire il piano e le sue stanze. Per ogni stanza, si vuole gestire l'illuminazione a scopo di risparmio energetico e utilizzare i display delle scrivanie per notificare informazioni utili. Si intendono realizzare, inoltre, funzionalità che coinvolgano tutte le stanze del piano, come ad esempio stabilire il numero totale di persone presenti.

Le funzionalità descritte sono realizzate attraverso agenti, detti *RoomAgent*, dedicati al controllo di ciascuna stanza e un agente *FlatAgent* per il controllo dell'intero piano (si veda Figura 7).

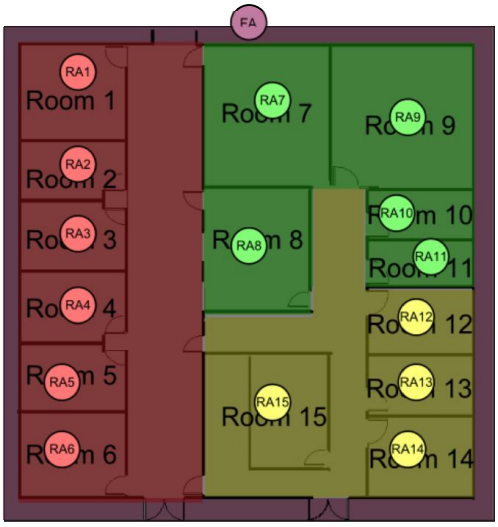

Figura 7: Distribuzione logica degli agenti.

Ogni *RoomAgent* ha la necessità di interagire con le risorse fisiche della stanza assegnata. E' conveniente, quindi, che esso risieda sull'Agent Server del nodo direttamente connesso a quelle risorse. L'agente del piano, invece, nell'ipotesi fatta, non ha la stessa necessità e, pertanto, può risiedere su un qualunque nodo, anche sul cloud.

4.3 Deployment dell'applicazione

Ogni applicazione parte da un nodo qualsiasi della rete dove è presente una componente preposta alla configurazione e avvio dell'applicazione. Di seguito ci

si riferirà a questa componente con il nome di *Deployer*. Sarà questa componente che invierà remotamente gli agenti verso la locazione opportuna (Figura 8).

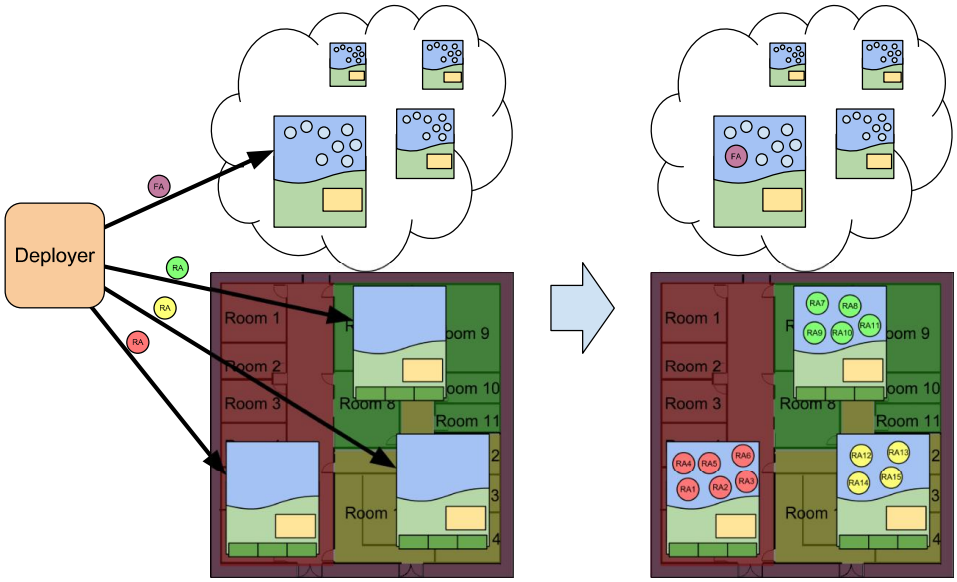

Figura 8: Deployment dell'applicazione.

4.4 Interazione tra gli agenti

I *RoomAgent* e il *FlatAgent* comunicano tra loro tramite scambio messaggi. In particolare, nel diagramma in Figura 9 è mostrata una possibile evoluzione delle interazioni tra gli agenti.

Dopo aver istanziato gli agenti sui vari nodi, il deployer invia a tutti i *RoomAgent* un messaggio con il quale fornisce loro la "*conoscenza*" del *FlatAgent*. Ogni *RoomAgent*, quindi, invia un messaggio al *FlatAgent* per notificargli la sua presenza. In seguito, ogni volta che una persona entra o esce da una stanza, il *RoomAgent* interessato notifica il numero di persone presenti nella "sua" stanza al *FlatAgent*, che provvederà ad aggiornare il conteggio delle persone presenti, al momento, nell'intero piano. Quando il *FlatAgent* verifica che vi è una sola persona nel piano, esso invia un messaggio al *RoomAgent* della stanza nella quale è presente la persona. In questo modo, il *RoomAgent* provvederà a mostrare l'informazione sul display delle scrivanie. Se il numero di persone aumenta, il *FlatAgent*

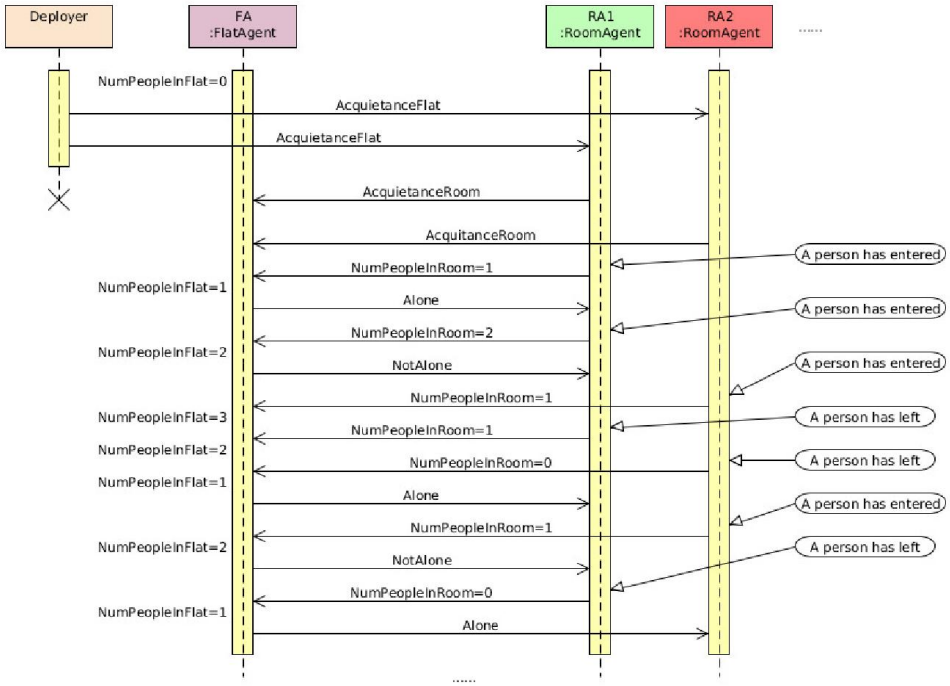

Figura 9: Esempio di interazione tra gli agenti.

notificherà al precedente *RoomAgent* che non vi è più una sola persona nel piano.

4.5 RoomAgent

Il *RoomAgent* è preposto al controllo di una singola stanza. Esso utilizza l'interfaccia esposta dal gateway di pertinenza per interagire con la parte fisica tramite i VO. Oltre a ciò, il *RoomAgent* è incaricato di scambiare informazioni d'interesse con il *FlatAgent* come visto nel paragrafo precedente.

Inizialmente il *RoomAgent* definisce le sue risorse d'interesse tramite il metodo *getResource* dell'interfaccia del gateway, ottenendo oggetti di tipo *VirtualResource* che utilizzerà durante il proseguio della sua esecuzione, come mostrato nel Codice 20 e 21.

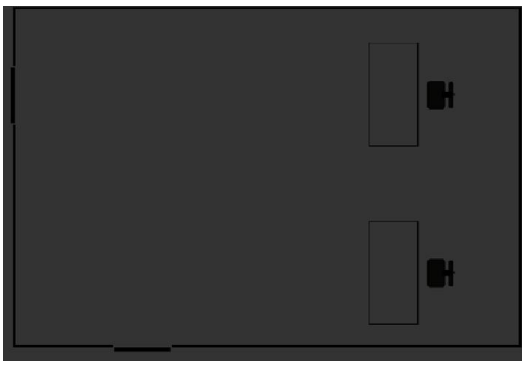

Figura 10: Stanza con porte chiuse.

Supponendo una stanza come quella mostrata in Figura 10, il controllo è effettuato secondo quanto riportato qui di seguito.

Comportamento I. Nel momento in cui una delle porte viene aperta, l'agente è notificato di tale evento e procede ad accendere tutte le luci relative ai 4 quadranti della stanza (Figura 11). Il Codice 15 presenta un frammento che realizza questo comportamento.

Comportamento II. Non appena è rilevato che una persona entra o esce dalla stanza, il *RoomAgent* invia un messaggio al *FlatAgent* contenente il numero di

Codice 15: RoomAgent, comportamento I.

```
/* inizialmente ci si sottoscrive agli eventi relativi alla
   apertura e chiusura delle porte */

gateway.subscribe(unlock_door0, this);
gateway.subscribe(unlock_door1, this);
gateway.subscribe(lock_door0, this);
gateway.subscribe(lock_door1, this);

...

/* nel metodo 'done' si riceve la notifica dell'evento relativa
   alla porta aperta, conseguentemente si accendono tutte le luci e
    si memorizza lo stato della porta */
if (ruleId.equals("unlock_door0")) {
      door0Opened = true;
      accendiTutteLeluci();
}

...

private void accendiTutteLeluci() {
      for (int i = 0; i < 4; i++) {
           accendi(i);
      }
}

private void accendi(int i) {
    switch (i) {
        case 0:
        // lightLevel mantiene lo stato delle luci
        if (lightLevel[0] < 1) {
            gateway.acting(add_light_wall0);
            lightLevel[0]++;
        }
        break;
        ...
    }
}
```

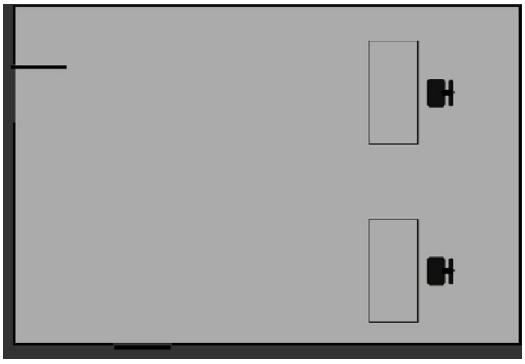

Figura 11: Stanza con porta aperta.

persone presenti, come descritto nella sezione 4.4. Nel Codice 16 si mostra un frammento che realizza questo comportamento (solo per l'ingresso di una persona).

Codice 16: RoomAgent, comportamento II.

```
/* si impostano le regole relative all'ingresso delle persone nella
     stanza, quindi ci si sottoscrive ad esse */
gateway.setRule(""+ingresso_door_0, "ingresso0");
gateway.setRule(""+ingresso_door_1, "ingresso1");
gateway.subscribe("ingresso0");
gateway.subscribe("ingresso1");

...

// nel metodo 'done'
if (ruleId.equals("ingresso0")||ruleId.equals("ingresso1")) {
    numPersone++;
    NumPeopleMsg npm = new NumPeopleMsg(numPersone);
    npm.setIdReceiver("FlatAgent");
    agentServer.send(npm, flatAgentServerHostPort);
}
```

Comportamento III. Se una persona entra in questa stanza e, in generale, se una persona si trova in uno dei 4 quadranti della stanza, la luce del quadrante relativo resta illuminata, mentre le luci nei rimanenti quadranti vengono spente per permettere un risparmio energetico. Quanto detto avviene solo se le porte risultano chiuse. Tale comportamento è mostrato in Figura 12 ed in Figura 13:

nella seconda figura la persona si è spostata di quadrante e le luci si sono adattate coerentemente. Nel Codice 17 si mostra un frammento che realizza il comportamento descritto.

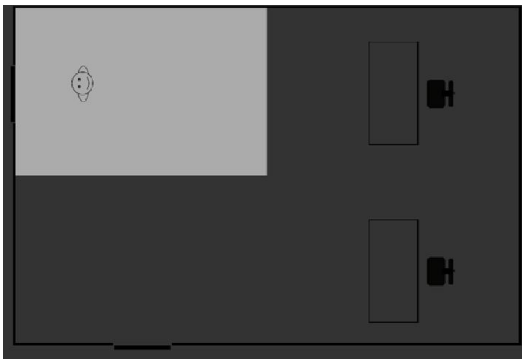

Figura 12: Stanza con persona nel primo quadrante.

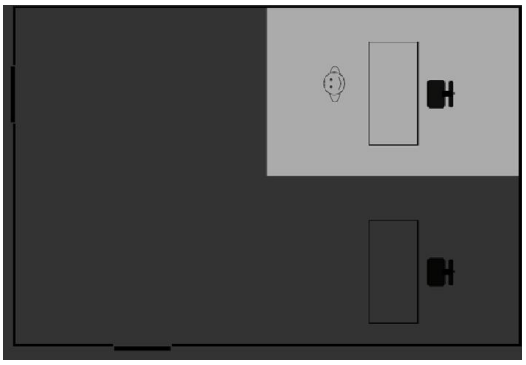

Figura 13: Stanza con persona nel secondo quadrante.

Comportamento IV. Quando una persona si siede ad una delle 2 sedie avvengono due attuazioni: la luce del quadrante relativo viene rafforzata e contemporaneamente sul display della Smart Desk la persona viene invitata a registrarsi (Figura 14). Il codice per realizzare questi comportamenti è analogo agli esempi precedenti.

Comportamento V. Il *RoomAgent*, quando riceve i messaggi da parte del *FlatAgent*, modificherà conseguentemente il testo mostrato sui display delle scri-

Codice 17: RoomAgent, comportamento III.

```
/* si impostano le regole relative alla presenza di persone nella
   stanza, quindi ci si sottoscrive ad esse */
gateway.setRule(""+num_persone_vicine_wall0+">0", "personaNear0");
gateway.setRule(""+num_persone_vicine_wall1+">0", "personaNear1");
gateway.setRule(""+num_persone_vicine_wall2+">0", "personaNear2");
gateway.setRule(""+num_persone_vicine_wall3+">0", "personaNear3");
gateway.subscribe("personaNear0");
gateway.subscribe("personaNear1");
gateway.subscribe("personaNear2");
gateway.subscribe("personaNear3");

...

/* nel metodo 'done' si riceve la notifica degli eventi precedenti,
    se le porte risultano chiuse si accende la luce nel quadrante
   relativo e si valuta se spegnere gli altri quadranti in base al
   loro stato */
if (ruleId.startsWith("personaNear") && !door0Opened && !
   door1Opened) {
      int i = Integer.parseInt(ruleId.substring(ruleId.length() -
         1));
      accendi(i);
      valutaAltriQuadranti(i);
}
```

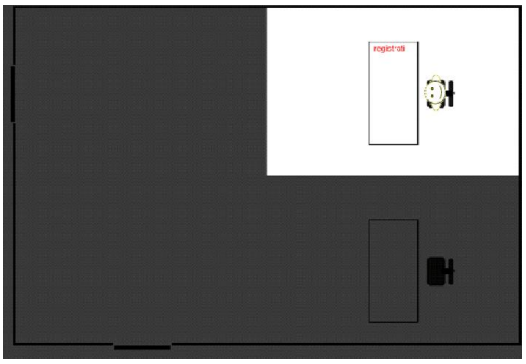

Figura 14: Stanza con persona seduta alla scrivania.

vanie. Il Codice 18 mostra un frammento che implementa tale comportamento.

Codice 18: RoomAgent, comportamento V.

```
public void receive(CPMessage msg) {
        ...
        if (msg instanceof AloneMsg)
            gateway.acting(display_desk0,"[text=you're alone!]");

        if (msg instanceof NotAloneMsg)
            gateway.acting(display_desk0,"[text=]");
        ...
}
```

4.6 FlatAgent

Il *FlatAgent*, come visto nella sezione precedente, dialoga con i *RoomAgent* mentre non utilizza VO (infatti può essere locato ovunque nella rete di nodi).

Nel Codice 19 un estratto che, alla ricezione di un messaggio contenente il numero di persone in una stanza, provvede ad effettuare il calcolo del numero totale di persone nel piano e, successivamente, se risulta una sola persona presente, invia un messaggio alla stanza opportuna.

Codice 19: FlatAgent, comportamento.

```
if (msg instanceof NumPeopleMsg) {
    updateCounter(msg);
    if (aloneInFlat())
        agentServer.send(new AloneMsg(), getRoomWithOnePerson());
}
```

Codice 20: RoomAgent, definizione delle risorse I.

```
VirtualResource prossimita_chair1 = gateway.getResource("
    prossimita_chair1","smart chair1","prossimita","[]");
VirtualResource prossimita_chair0 = gateway.getResource("
    prossimita_chair0","smart chair0","prossimita","[]");
VirtualResource num_persone_vicine_desk1 = gateway.getResource("
    num_persone_vicine_desk1","smart desk1","num_persone_vicine","[]
    ");
VirtualResource num_persone_vicine_desk0 = gateway.getResource("
    num_persone_vicine_desk0","smart desk0","num_persone_vicine","[]
    ");
VirtualResource lock_door1 = gateway.getResource("lock_door1","
    smart door1","lock","[]");
VirtualResource lock_door0 = gateway.getResource("lock_door0","
    smart door0","lock","[]");
VirtualResource display_desk1 = gateway.getResource("display_desk1"
    ,"smart desk1","display","[]");
VirtualResource display_desk0 = gateway.getResource("display_desk0"
    ,"smart desk0","display","[]");
VirtualResource unlock_door1 = gateway.getResource("unlock_door1","
    smart door1","unlock","[]");
VirtualResource unlock_door0 = gateway.getResource("unlock_door0","
    smart door0","unlock","[]");
VirtualResource less_light_wall3 = gateway.getResource("
    less_light_wall3","smart wall","less_light","[zona =3]");
VirtualResource uscita_door1 = gateway.getResource("uscita_door1","
    smart door1","uscita","[]");
VirtualResource less_light_wall2 = gateway.getResource("
    less_light_wall2","smart wall","less_light","[zona =2]");
VirtualResource uscita_door0 = gateway.getResource("uscita_door0","
    smart door0","uscita","[]");
VirtualResource less_light_wall1 = gateway.getResource("
    less_light_wall1","smart wall","less_light","[zona =1]");
VirtualResource less_light_wall0 = gateway.getResource("
    less_light_wall0","smart wall","less_light","[zona =0]");
VirtualResource utilizzazione_sedia_chair1 = gateway.getResource("
    utilizzazione_sedia_chair1","smart chair1","utilizzazione_sedia"
    ,"[]");
```

Codice 21: RoomAgent, definizione delle risorse II.

```
VirtualResource utilizzazione_sedia_chair0 = gateway.getResource("
    utilizzazione_sedia_chair0","smart chair0","utilizzazione_sedia"
    ,"[]");
VirtualResource light_off_wall3 = gateway.getResource("
    light_off_wall3","smart wall","light_off","[zona =3]");
VirtualResource light_off_wall2 = gateway.getResource("
    light_off_wall2","smart wall","light_off","[zona =2]");
VirtualResource light_off_wall1 = gateway.getResource("
    light_off_wall1","smart wall","light_off","[zona =1]");
VirtualResource add_light_wall3 = gateway.getResource("
    add_light_wall3","smart wall","add_light","[zona =3]");
VirtualResource light_off_wall0 = gateway.getResource("
    light_off_wall0","smart wall","light_off","[zona =0]");
VirtualResource add_light_wall2 = gateway.getResource("
    add_light_wall2","smart wall","add_light","[zona =2]");
VirtualResource add_light_wall1 = gateway.getResource("
    add_light_wall1","smart wall","add_light","[zona =1]");
VirtualResource add_light_wall0 = gateway.getResource("
    add_light_wall0","smart wall","add_light","[zona =0]");
VirtualResource ingresso_door1 = gateway.getResource("
    ingresso_door1","smart door1","ingresso","[]");
VirtualResource ingresso_door0 = gateway.getResource("
    ingresso_door0","smart door0","ingresso","[]");
VirtualResource num_persone_vicine_wall3 = gateway.getResource("
    num_persone_vicine_wall3","smart wall","num_persone_vicine3","[
    zona =3]");
VirtualResource num_persone_vicine_wall2 = gateway.getResource("
    num_persone_vicine_wall2","smart wall","num_persone_vicine2","[
    zona =2]");
VirtualResource num_persone_vicine_wall1 = gateway.getResource("
    num_persone_vicine_wall1","smart wall","num_persone_vicine1","[
    zona =1]");
VirtualResource num_persone_vicine_wall0 = gateway.getResource("
    num_persone_vicine_wall0","smart wall","num_persone_vicine0","[
    zona =0]");
```

5 Primi esperimenti con RaspberryPi e Arduino

Come già visto in precedenza, sia la piattaforma ad agenti che quella per la virtualizzazione degli oggetti sono fisicamente poste su dei nodi di calcolo, sparsi per l'ambiente da monitorare e controllare. A tale scopo, in una prima implementazione di test, si è pensato di utilizzare dei computer leggeri, economici e performanti quali sono i Raspberry Pi. L'economicità dei costi di questi dispositivi rende possibile pensare di installarne una certa moltitudine nell'ambiente gestito dalla piattaforma CPS, e la loro programmabilità general pupose (sono in effetti dei pc arm sui quali può girare una distibuzione linux) li rende validi sia per obiettivi di sviluppo che di implementazine sul campo.

La disponibilità sul mercato di hardware come RaspberryPi rappresenta, in realtà, una tecnologia abilitante del framework oggetto del documento. In ambito cyber-physical si utilizzano generalmente architetture a logica cablata per il monitoraggio/controllo della parte fisica, per rispondere ai requisiti di costo, consumo energetico e performance. Hardware come i RaspberryPi risponde ai medesimi requisiti offrendo, inoltre, la programmabilità general purpose.
Per quanto riguarda il comparto sensori/attuatori, l'attuale implementazione fisica prevede l'utilizzo di dispositivi Arduino. Tali dispositivi sono costituiti da un microcontrollore programmabile, che può essere collegato a diversi altre componenti elettroniche, nel nostro caso specifici sensori e attuatori fisici. In generale, l'architettura proposta si adatta ad essere utilizzata anche con altri dispositivi, quali, ad esempio, i MICAz.

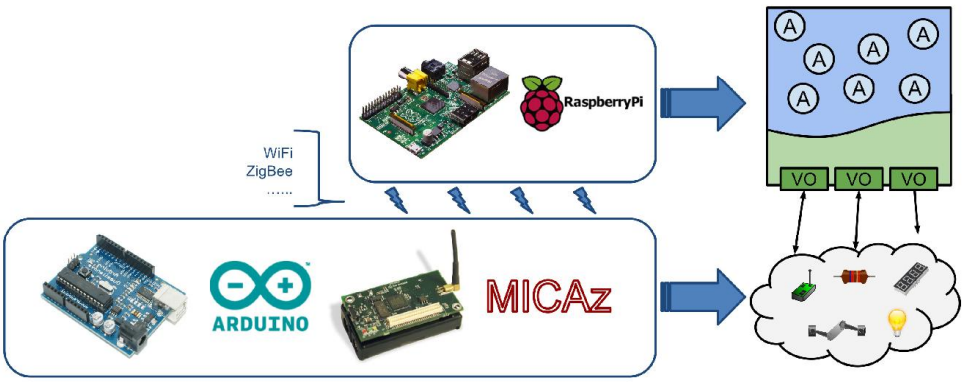

Figura 15: Esempi dispositivi utilizzabili.

6 Bibliografia

[for13] G. Fortino, A. Guerrieri, M. Lacopo, M. Lucia, W. Russo. "An Agent-Based Middleware for Cooperating Smart Object". In PAAMS 2013 Workshop, CCIS 365, Springer, 2013.

[hua08] B. X. Huang. Cyber Physical Systems: A survey. Presentation Report, Giugno 2008.

[jes13] S. Jeschke, "Cyber-physical systems - History, Presence and Future". Industrial Advisory Board,

[kar11] S. Karnouskos. Cyber-physical systems in the SmartGrid. Industrial Informatics (INDIN), 2011 9th IEEE International Conference on. IEEE, 2011.

[kou09] A. Koubaa, B. Andersson, "A vision of cyber-physical internet". In proc. Of the Workshop of Real-Time Networks (RTN 2009), Satellite Workshop to (ECRTS 2009).

[lee06] E.A. Lee Cyber-physical systems-are computing foundations adequate.Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap. Vol. 2. 2006.

[lei13] P. Leitão. "Towards Self-organized Service-Oriented Multi-agent Systems". In Studies in Computational Intelligence Volume 472 2013, Springer.

[lin10] J. Lin, S. Sedigh, A. Miller. "Modeling Cyber-Physical Systems with Semantic Agents". In Computer Software and Application Conference Workshops (COMPSACW), IEEE 2010.

[nist13] Autori vari, "Foundation for Innvoation in Cyber-Physical Systems", 2013. Aachen, Febbraio 2013.

[san12] T. Sanislav, L. Miclea. "Cyber-physical systems - Concept, Challanges and Research Areas". In Control Engineering and Applied Informatics, Vol.14, No.2, pp. 28-33, 2012.

[sci11] J. Shi, J. Wan, H. Yun, H. Suo, "A Survey of Cyber-Physical Systems". In proc. Of the Int. Conf. On Wireless Communications and signal Processing, Nanjing, China, November 9-11, 2011.

[tal08] C.L. Talcott. Cyber-Physical Systems and Events. Software-Intensive Systems and New Computing Paradigms 2008: 101-115.