UNIVERSITÀ
DEGLI STUDI
DI PALERMO

NDS
NETWORKING AND DISTRIBUTED SYSTEMS
RESEARCH GROUP

Integrazione di dispositivi wearable e fissi per l'health monitoring

Tesi di Laurea Magistrale in Ingegneria Informatica

Riccardo Rizzo

Relatore: Prof. Daniele Peri

UNIVERSITÀ DEGLI STUDI DI PALERMO

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

INTEGRAZIONE DI DISPOSITIVI WEARABLE E FISSI PER L'HEALTH MONITORING

TESI DI LAUREA DI

RICCARDO RIZZO

RELATORE

Prof. DANIELE PERI

CONTRORELATORE

Prof. GIUSEPPE LO RE

ANNO ACCADEMICO 2019 – 2020

Sommario

Premessa

Questa tesi ha come obiettivo quello di porre le basi attraverso le quali sarà possibile costruire un sistema domotico capace di fornire supporto a persone bisognose di assistenza, ma allo stesso tempo desiderose di condurre la propria vita in autonomia.

La tesi esporrà in primis una sezione riguardante lo stato dell'arte esplorando quindi gli ambienti intelligenti e le tecnologie di cui attualmente si dispone in commercio per realizzarli. Si introdurrà successivamente il concetto di monitoraggio della salute e si analizzeranno le metodologie più efficienti per instaurare una comunicazione tra i vari nodi sensori di una rete. Ci si soffermerà sul concetto di applicazione distribuita e verrà mostrato il modo in cui un'applicazione distribuita può operare in contesti domestici, affrontando in particolare le problematiche legate alla comunicazione di eventi in tempo reale, con particolare attenzione al rilevamento delle cadute.

Poiché spesso testare sistemi di questo tipo potrebbe non essere semplice in contesti di sviluppo, è stato costruito un meccanismo capace di virtualizzare sensori mediante l'utilizzo di dati precampionati raccolti all'interno di dataset. Si passerà quindi all'analisi dei dati ottenuti e delle prestazioni del sistema realizzato. Per concludere, sono stati esposti i possibili sviluppi futuri che potranno apportare delle migliorie prestazionali al sistema.

1 Introduzione

Ai giorni d'oggi la tecnologia ricopre un ruolo chiave nella vita quotidiana di ogni individuo e in un periodo di grande evoluzione come quello che si sta attraversando sarebbe impensabile farne a meno. Il progresso tecnologico ad esempio ha reso possibile l'utilizzo del proprio smartphone per gli scopi più disparati, dall'acquisto online di prodotti con un semplice click all'interazione con gli impianti elettrici della propria abitazione anche per gli utenti meno esperti. A tal proposito è possibile introdurre il concetto di *Smart Environment*. Uno *Smart Environment* può essere identificato come un ambiente all'interno del quale sono presenti degli strumenti di monitoraggio che cooperarando in sincronia permettono di tenere sotto controllo aspetti critici in relazione a quel particolare ambiente [1]. In una piscina ad esempio è necessario monitorare parametri come il livello di cloro e il pH dell'acqua. Un primo passo per rendere *smart* un ambiente di questo tipo potrebbe essere quello di avere dei sensori che monitorino costantemente le percentuali di tali parametri e, magari tramite un'applicazione mobile, inviino delle notifiche push nel momento in cui risultano fuori norma. Per rendere un ambiente di questo tipo ancora più *smart* è possibile aggiungere degli attuatori che agiscono in base ai dati rilevati dai sensori. Il possessore della piscina in questo caso non dovrà preoccuparsi ad esempio di aggiungere del cloro perché troppo basso, ma sarà direttamente lo *smart environment* a pensarci per lui.

Dal momento in cui l'automatizzazione è all'ordine del giorno, di ambienti intelligenti se ne possono immaginare molteplici e uno dei più grandi potrebbe essere identificato in quelle che sono chiamate *Smart Cities*, ovvero città in cui l'utilizzo di tecnologie digitali migliora

significativamente l'utilizzo dei servizi pubblici, massimizzandone l'efficienza.

Se al giorno d'oggi gli *Smart Environment* si sono estesi a tal punto da coprire intere città, non poteva essere altrimenti per le singole abitazioni. In particolare negli ultimi anni, grazie anche allo sviluppo dell'intelligenza artificiale [2], il concetto di casa intelligente sta prendendo sempre più piede e i dispositivi in commercio da integrare negli ambienti domestici sono in numero sempre crescente. A tal proposito è bene introdurre il concetto di IoT (*Internet of Things)*. Il termine si riferisce appunto all'internet delle cose, ovvero al modo in cui l'evoluzione di internet ha permesso la sua integrazione con oggetti con cui ognuno di noi ha a che fare durante la propria quotidianità. Gli oggetti che stanno alla base dell'IoT sono generalmente quelli che vengono chiamati *smart objects*, ovvero dispositivi intelligenti che hanno anche una minima capacità di elaborazione e la possibilità d'interazione con l'ambiente che li circonda. Ai giorni nostri i campi di applicazione dell'IoT sono veramente tanti e negli anni a venire lo sviluppo del settore fornirà sicuramente una migliore qualità della vita e dei servizi che le città mettono a disposizione.

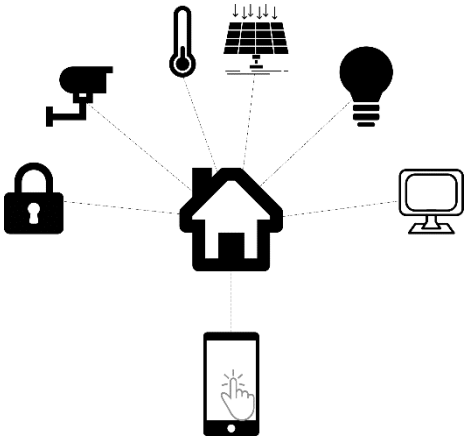

Figura 1 – Struttura generale di una smart home

1.1 Cenni sulla Domotica

Il termine domotica nasce dal neologismo francese *domotique,* espressione che racchiude due parole: la prima, *domus,* proviene dal latino e significa proprio *casa*, la seconda, deriva dal francese *telematique* e riguarda l'ambito delle telecomunicazioni. La domotica può abbracciare diverse discipline quali l'ingegneria civile, l'elettrotecnica, l'elettronica, il settore informatico e quello delle telecomunicazioni. Spesso una *smart house* viene associata ad un bene di lusso, ma più avanti andremo negli anni, maggiore sarà l'accessibilità ai prodotti *smart* da parte di tutti e la tendenza a rendere intelligenti le nostre abitazioni verrà sempre più. Oltre a migliorare la qualità e la sicurezza della propria vita, un ambiente smart permette un monitoraggio accurato degli impianti casalinghi e un risparmio notevole di energia e costi di gestione.

La rivoluzione informatica è stata capace di modificare i processi produttivi all'interno delle aziende, rendendo possibile l'automazione in campo manufatturiero mediante l'integrazione dei sistemi di automazione industriale.

Il primo sistema domotico venne realizzato nel 1966 da un ingegnere statunitense, ma solo dopo i primi anni novanta il settore cominciò ad affermarsi sempre di più, grazie anche alla nascita di sistemi e protocolli sempre più standardizzati.

Inizialmente l'*home automation*, prevedeva un sistema di tipo centralizzato (Figura 2) in cui ogni settore (per settore s'intende ad esempio quello elettrico, quello della sicurezza o il settore *HVAC "Heating Ventilation Air Conditioning"*) era indipendente dagli altri. Il problema di un sistema di questo tipo risiedeva nel fatto che era possibile una ridondanza di

dispositivi connessi la cui utilità era richiesta in settori differenti, ma in particolar modo l'impossibilità tra i vari settori di cooperare per coordinare le proprie attività. Dal punto di vista dei costi, un sistema del genere risultava eccessivamente dispendioso. Essendo infatti un sistema chiuso e inestensibile, la possibilità di effettuare modifiche era inesistente e sebbene le performace complessive potevano definirsi soddisfacenti, col passare del tempo l'intero sistema sarebbe diventato obsoleto.

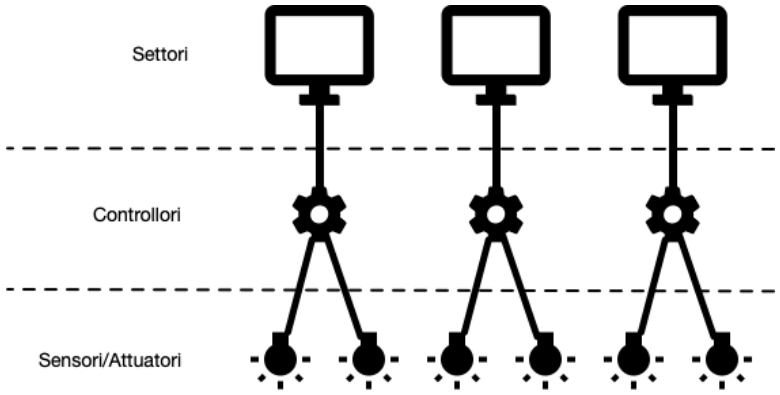

Figura 2 – Sistema domotico centralizzato

Con l'avvento dei sistemi a BUS, dell'architettura di tipo Client/Server e di protocolli standardizzati ed aperti, è stato possibile realizzare un sistema di tipo decentralizzato (Figura 3) in cui la totalità dell'ambiente era interconnessa. Un'architettura di questo tipo permette una grande flessibilità garantendo una scalabilità del sistema non indifferente, nonché la possibilità di connettere alla rete i vari sensori ed eliminare le ridondanze condividendo tra i vari settori i moduli necessari.

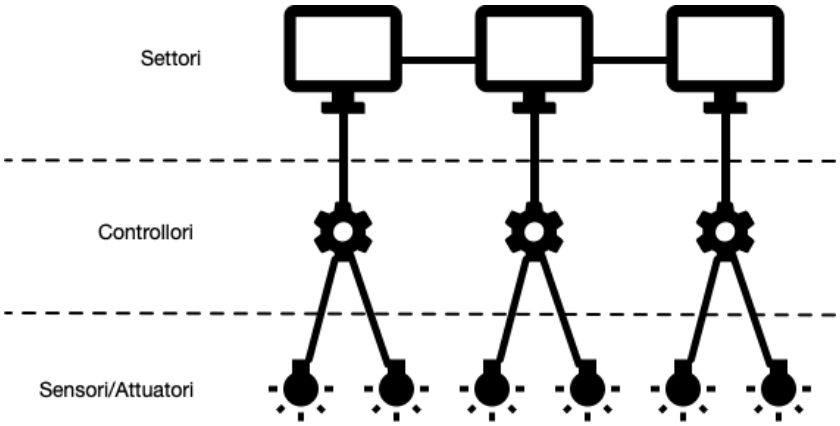

Figura 3 - Sistema domotico decentralizzato

1.2 La domotica oggi

La tendenza degli ultimi anni si contrappone del tutto a ciò che veniva realizzato in precedenza. Se prima si aveva a che fare con sistemi chiusi e inestensibili, adesso si tende a produrre infrastrutture con tecnologia *open hardware* la quale non solo presenta dei costi nettamente inferiori, ma permette all'acquirente di personalizzare ed estendere il proprio ambiente domotico come meglio crede. A tal proposito sono stati sviluppati e resi standard una serie di protocolli sui quali ormai si basano la quasi totalità dei dispositivi *smart* prodotti odiernamente. Sistemi centrali ormai ampiamente diffusi come *Google Home* o *Amazon Alexa* (Figura 4) risultano accessibili dal punto di vista economico. In aggiunta è possibile

Figura 4 – Due diverse implementazioni di assistenti vocali, sulla sinistra quella di Google, sulla destra quella di Amazon

acquistare moduli *smart* opzionali interfacciabili col sistema centrale senza la necessità di una configurazione.

In questo modo il costo del sistema domotico nel suo complesso, crescerà al crescere del numero di moduli aggiuntivi, e un utente che ha la semplice necessità di accendere o spegnere le luci da remoto, o di attivare il condizionatore prima di rientrare a casa in una giornata particolarmente calda, non ha bisogno di spendere una fortuna.

Fino ad ora si è discusso di domotica in termini abbastanza generici sottolineando principalmente i benefici di vivere in una *smart home* principalmente dal punto di vista del comfort. Si immaginino invece i benefici che soggetti bisognosi d'assistenza potrebbero trarre conducendo la propria vita all'interno di un ambiente intelligente. La tesi in questione ha l'obiettivo di porre le basi per un sistema domotico indirizzato a persone bisognose d'assistenza, ma che al contempo desiderano di vivere la propria vita in autonomia, senza dover dipendere troppo da qualcuno. La casa intelligente di una persona anziana, ad esempio, dovrebbe essere in grado di prevenire situazioni di pericolo legate sia alla salute fisica dell'assistito, che all'ambiente che lo circonda. Il sistema dovrebbe ad esempio essere in grado di avvisare tempestivamente il *caregiver* nel caso in cui in casa sia stata rilevata la presenza di fumo o gas, oppure se il battito cardiaco del soggetto si presenta insolitamente elevato. Per *caregiver* s'intende la persona che presta le proprie cure all'assistito. Generalmente tale figura è rappresentata da un componente facente parte del nucleo familiare del paziente. La tempestività con la quale il sistema avverte chi di dovere potrebbe evitare spiacevoli inconvenienti e addirittura, nei casi più critici, salvare la vita.

Prima di addentrarci su questo aspetto, vediamo una carrellata di dispositivi che potrebbero interfacciarsi con infrastrutture già esistenti come *Google Home* o *Alexa* già citate in precedenza.

Di dispositivi smart compatibili con tali infrastrutture, ne esistono a bizzeffe.

Uno dei più ovvi è sicuramente la semplice lampadina (Figura 5), la quale se prima poteva essere soltanto accesa o spenta tramite uno switch, adesso è possibile controllarla con la propria voce gestendone l'accensione, lo spegnimento, cambiarne il colore o addirittura raggrupparla assieme ad altre per far sì che possano essere controllate contemporaneamente.

Figura 5 - Lampadina smart

Nella maggior parte dei casi, non tutti gli elettrodomestici all'interno di un'abitazione sono *smart*. Per sopperire a tale inconveniente è possibile alimentare quegli elettrodomestici tramite prese intelligenti (Figura 6), utilizzabili ovviamente anche tramite interazione vocale.

Figura 6 - Presa smart

Oltre prese e lampadine possiamo trovare in commercio anche interruttori WiFi, termostati, impianti di riscaldamento, videocamere di sorveglianza, robot pulitori, dispositivi audio e *smart TV*. Probabilmente col passare degli anni le novità in questo

campo saranno all'ordine del giorno e i dispositivi compatibili con le infrastrutture già esistenti saranno in numero sempre maggiore. Ciò ha scatenato negli utenti l'interesse nell'interagire con tali dispositivi in modo personalizzabile. Particolarmente interessante è IFTTT (*If This Than That*). IFTTT è un servizio online gratuito che permette la creazione dei cosiddetti applet, ovvero semplici costrutti di tipo *trigger-action* che permettono di effettuare una determinata azione quando un evento ben preciso si scatena. IFTTT permette ad esempio di accendere la luce di una stanza ogni giorno dopo il tramonto in modo automatizzato.

1.3 Health Monitoring

Per *health monitoring* si intende un insieme di sistemi e risorse atti a monitorare lo stato di salute di un individuo. Fino a non molti anni fa, questo tipo di pratica veniva svolta soltanto nelle strutture sanitarie, come ad esempio gli ospedali o le cliniche private, ed era richiesto un personale esperto in grado di coordinare il funzionamento delle apparecchiature, oltre che capace di interpretare il flusso di dati clinici proveniente da queste ultime. Dal punto di vista monetario e spazio fisico occupato dalle apparecchiature, gli enti sanitari dispongono sicuramente di fondi e spazi fisici che in un contesto domestico non sarebbe possibile avere.

Il costo dei macchinari necessari per effettuare *health monitoring* è infatti piuttosto elevato e soltanto enti come ospedali o cliniche possono permettersi spese sostanziose per apparecchiature di questo tipo. Inoltre, spesse volte, il volume fisico occupato da queste apparecchiature è piuttosto notevole, ragion per cui non sono adatte a contesti casalinghi.

Negli ultimi anni si è sviluppata una forte tendenza nel cercare di portare dispositivi simili all'interno delle mura domestiche, cosicché le persone possano monitorare la propria salute in maniera del tutto autonoma, senza dover necessariamente recarsi presso una struttura sanitaria. Naturalmente le prestazioni, l'efficienza e soprattutto l'affidabilità di tali dispositivi non sono paragonabili a quelli utilizzati in ambito ospedaliero. L'obiettivo infatti non è in alcun modo quello di sostituire completamente questi ultimi, ma di realizzare dispositivi facilmente accessibili e configurabili da persone che non dispongono di competenze mediche.

Il primo problema da affrontare è di tipo spaziale, ovvero quello di concentrare tale tecnologia in dispositivi sufficientemente piccoli in modo da essere facilmente installati in una struttura domestica o addirittura indossati dall'individuo, limitando al minimo l'ingombro o il fastidio che essi possono costituire. Il secondo, invece, è quello di sintetizzare e isolare le funzionalità principali, ovvero capire quali dati possono realmente essere letti dall'apparecchio con una sufficiente approssimazione. È chiaro che in un contesto domestico non sarà possibile monitorare parametri clinici come ad esempio quelli sanguigni. Le analisi del sangue richiedono infatti qualcuno che effettui un prelievo, quindi macchinari e reagenti appositi, nonché la consultazione dei risultati da parte di un medico esperto.

Nel corso degli anni sono stati sviluppati sistemi per *health monitoring* più o meno complessi a seconda delle esigenze degli individui a cui erano destinati. Nei casi più semplici si trovano dispositivi indossabili destinati ad un pubblico di massa, basti pensare all'avvento degli smartwatch e delle smartband, che sono in grado di monitorare il battito cardiaco. Chiaramente tali dispositivi sono destinati ad un uso sportivo o di puro appagamento personale e non possono sicuramente essere catalogati come dispositivi

medici. Per persone che necessitano invece di assistenza medica quotidiana sono state sviluppate delle vere e proprie infrastrutture che prevedono l'installazione di numerosi sensori di vario tipo all'interno dell'abitazione, coordinati da un sistema centrale. In questo caso si ha a che fare con sistemi più complessi che, a differenza di uno smartwatch che si indossa al polso ed è pronto all'uso, necessitano di una fase di installazione e configurazione. Esistono inoltre sistemi ibridi in cui l'infrastruttura domotica comunica con un dispositivo indossato dall'utente scambiando informazioni circa i parametri vitali dell'individuo. È proprio questo l'argomento centrale su cui si concentrerà questa tesi.

2 Sistemi distribuiti

Come accennato precedentemente, il sistema completo permetterà di effettuare *health monitoring* e *home assistant* in un contesto in cui il soggetto target è una persona che richiede assistenza domestica. Dall'altra parte del sistema è presente un *caregiver* che deve essere opportunamente avvisato in caso di situazioni di emergenza. Si presenta di seguito uno schema generale (Figura 7) che possa aiutare il lettore ad avere un'idea del funzionamento generale del sistema voluto.

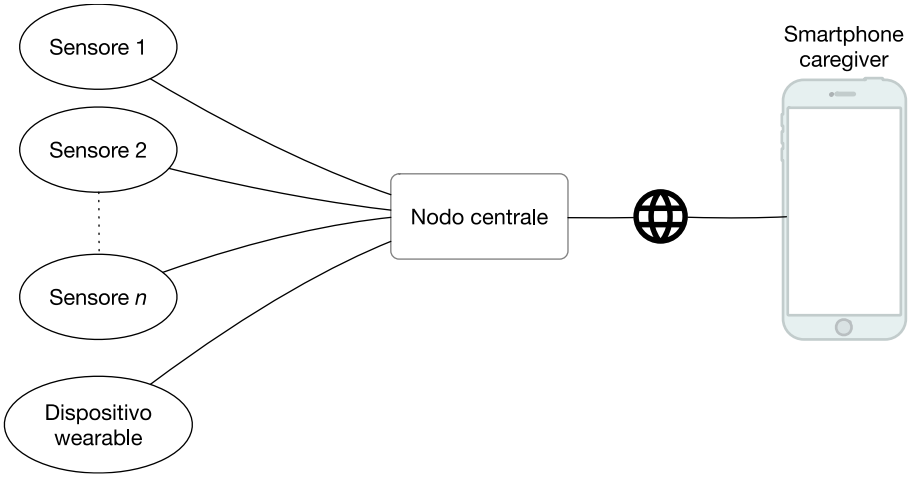

Figura 7 - Schema generale di un ipotetico sistema completo

Parleremo più avanti nel dettaglio di un'infrastruttura come quella mostrata in figura.

2.1 Applicazioni Distribuite

Un sistema distribuito va inteso come un sistema informatico composto da un insieme di processi che, scambiandosi messaggi, cooperano per raggiungere un fine comune e appaiono all'utilizzatore come un unico grande processo. Lo strato software tramite il quale avviene la comunicazione tra processi differenti prende il nome di *middleware*. Dato quindi un sistema distribuito, possiamo definire applicazione distribuita quel processo che sfrutta i vari nodi del sistema per elaborare ed eventualmente conservare le informazioni di interesse. Un sistema distribuito è generalmente accessibile in modalità remota e dovrebbe garantire il corretto funzionamento anche se uno dei nodi presenta dei malfunzionamenti. L'assenza di un clock globale rende impossibile effettuare una perfetta sincronizzazione e a causa di questo non è possibile definire con certezza assoluta lo stato di un sistema distribuito. Una caratteristica di sistemi di questo tipo è senza dubbio l'eterogeneità. Nessuno vieta infatti che i nodi che lo compongono, si differenzino per componenti hardware o addirittura per il software.

2.1.1 Applicazioni distribuite per smart environment

Le applicazioni di cui abbiamo parlato finora fanno pensare a grandi sistemi con elevate capacità di calcolo. Tuttavia è possibile gestire applicazioni distribuite su piccola scala, ad esempio all'interno di una rete locale, in cui i nodi sono rappresentati da piccole macchine dalle risorse

limitate interconnesse tra loro mediante un nodo centrale [3]. Si pensi ad esempio ad un ambiente domotico in cui è presente un'unità centrale alla quale sono connessi vari nodi sensori smart. In ambienti così fatti, è possibile pensare ad ogni nodo sensore come un piccolo calcolatore che esegue le proprie elaborazioni e, se necessario, comunica al nodo centrale i risultati. Supponiamo ad esempio di avere un nodo centrale al quale sono connessi due sensori smart: uno per il rilevamento del fumo, l'altro per il gas. Nei due sensori probabilmente girerà una routine che si occuperà di verificare costantemente la presenza di fumo o gas e soltanto quando ciò avverrà, ci sarà la comunicazione col dispositivo centrale, il quale si occuperà di prendere le contromisure necessarie, come attivare il sistema antincendio o chiamare i soccorsi. Questo, seppur semplice, è un esempio di applicazione distribuita per smart environment.

Applicazioni di questo tipo vedono spesso come protagonisti nodi con capacità di calcolo limitate che scambiano tra loro informazioni più o meno complesse. In generale tali dispositivi non dispongono di un supporto per implementazioni avanzate e la latenza dovuta allo scambio dei dati potrebbe non essere trascurabile. La riprogrammazione dei nodi inoltre è necessaria nel momento in cui le funzionalità del singolo nodo vogliono essere modificate. Date le risorse limitate dei nodi sarebbe impensabile far girare negli stessi gli interpreti dei linguaggi più moderni come Java o Python poiché le risorse impiegate sarebbero considerevoli. A tal proposito un esempio di middleware per applicazioni distribuite che è bene citare è DC4CD [4] [5] [6], progettato per consentire lo scambio di codice eseguibile tra dispositivi con risorse di calcolo limitate. L'interprete del codice in questo caso non dispone di un'architettura complessa e stratificata, al contrario, ha una corrispondenza molto vicina con il linguaggio macchina.

Un interprete che dispone di caratteristiche come quelle descritte potrebbe essere quello di Forth. Forth è un linguaggio semplice e flessibile che permette di scrivere del codice compatto e facilmente interpretabile non solo dalla macchina, ma anche dall'uomo. L'interprete Forth è molto leggero e per questo si presta bene per risiedere all'interno di dispositivi dalle risorse limitate. DC4CD pone come obiettivo lo scambio di codice simbolico superando il problema della riprogrammazione dei nodi [7] e pone le basi per l'implementazione di applicazioni complesse anche su dispositivi dalle risorse limitate.

2.1.2 Applicazioni distribuite per health monitoring

Si è già parlato di applicazioni distribuite e si è visto come queste possano essere utilizzate per svariati scopi: dal puro appagamento personale di avere una casa totalmente automatizzata fino a scopi di sicurezza domestica, come la prevenzione e l'intervento in caso di situazioni di emergenza. Un altro campo per cui è possibile scrivere applicazioni distribuite è quello dell'*health monitoring*. Il settore dell'IoT in campo medico è in continuo avanzamento, e allo stato attuale è già possibile trovare dispositivi indossabili con a bordo un gran numero di sensori in uno spazio ridotto, capaci di raccogliere informazioni sullo stato di salute del paziente e anticipare situazioni critiche molto prima che esse possano verificarsi. Il continuo sviluppo dell'intelligenza artificiale permetterà sempre più di raccogliere dati significativi che potranno essere utilizzati come base di conoscenza per applicazioni di monitoraggio della salute. Come le applicazioni distribuite per *smart environment* prevedono

sensori ambientali cooperanti tra loro, vale lo stesso per applicazioni distribuite per il monitoraggio della salute. In questo caso il paziente indosserà una serie di sensori ognuno con uno scopo ben preciso, capaci probabilmente di comunicare con infrastrutture esterne poste all'interno di ospedali o cliniche. Si parla in questo casi di *body area network* [8] In questo modo i medici saranno in grado di monitorare lo stato di salute dei propri pazienti dall'esterno ed interagire in casi di emergenza. Si parlerà più in dettaglio di tale argomento nella tesi complementare a questa.

2.1.3 Applicazione distribuita per il rilevamento di eventi

Applicazioni distribuite per il rilevamento di eventi in *real time* sono di fondamentale importanza nel momento in cui si vuole realizzare un sistema capace di rispondere a determinate situazioni in modo tempestivo. Si è già parlato precedentemente di applicazioni distribuite per smart environment e si è fatto l'esempio dei sensori smart di fumo e gas. Un'applicazione di quel tipo può catalogarsi oltre che come applicazione distribuita per smart environment, anche come applicazione distribuita per il rilevamento di eventi, poiché il rilevamento dell'evento generato dalla presenza di fumo o gas genera un messaggio di allarme che sarà opportunamente elaborato dal nodo centrale.

Durante lo sviluppo della tesi in esame, ci si concentrerà particolarmente sulle applicazioni distribuite per il rilevamento di eventi. Come già accennato, lo scopo futuro è quello di costruire un sistema a cui persone bisognose d'assistenza potranno fare affidamento per la segnalazione di situazioni di emergenza in modo tempestivo. Il concetto di evento in questo caso è fondamentale. In particolare possiamo distinguere due classi di

eventi: quelli legati allo stato fisico dell'assistito e quelli invece legati alla sicurezza dell'ambiente che lo circonda.

Nella Figura 8 sottostante è possibile osservare uno scenario molto generico all'interno del quale un'applicazione distribuita per il rilevamento di eventi opera.

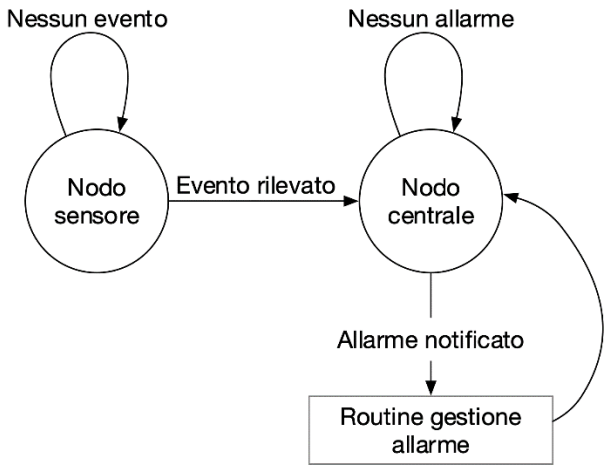

Figura 8 - Schema di funzionamento per l'interconnessione tra un nodo sensore ed un nodo centrale

2.2 WSN – Wireless Sensor Network

Uno scenario come quello appena visto è ovviamente estensibile aggiungendo più nodi sensori. Una topologia costituita da un nodo centrale e da più nodi sensori wireless cooperanti prende il nome di WSN (Wireless Sensor Network) [9]. Una rete di sensori generalmente opera in ambienti circoscritti all'interno del fenomeno che si vuole osservare. Se si vuole creare un sistema di assistenza domestica smart, è chiaro che la rete di sensori si troverà all'interno dell'abitazione target. I dispositivi che vengono adoperati come sensori all'interno delle WSN hanno in genere peso,

dimensioni e consumo energetico trascurabile e il canale di trasmissione tramite il quale avviene la comunicazione è ovviamente wireless. In genere le WSN sono molto utilizzate per la raccolta dei dati i quali dopo essere stati elaborati dal nodo centrale possono essere inviati ad un servizio di Cloud. Per ragioni di privacy e sicurezza non è però necessario che tutti i dati raccolti vengano messi in rete, o quantomeno è necessario analizzare per bene gli scopi a cui un'applicazione distribuita è destinata e stabilire cosa è opportuno memorizzare in locale e cosa invece può transitare tramite internet. Talvolta i dati grezzi raccolti dai nodi sensori potrebbero addirittura non essere inviati al nodo centrale. I piccoli processori situati a bordo di essi sono infatti in grado di effettuare piccole porzioni di elaborazioni tali da restituire dei dati già pronti per elaborazioni successive. In questo modo il carico di lavoro da affidare al nodo centrale è minore, e vengono sfruttate al meglio le potenzialità di una WSN, specie nei casi in cui i nodi sensori sono tanti e l'elaborazione di troppe informazioni risulterebbe onerosa per il nodo principale.

La progettazione di una rete di sensori richiede la valutazione di importanti aspetti. Il primo è la cosiddetta *fault-tolerance* e indica la capacità di un sistema di funzionare correttamente anche nel caso in cui uno o più nodi sensori presentino malfunzionamenti. Più è critica l'applicazione per WSN da realizzare, maggiore deve essere la tolleranza ai guasti. Un altro importante aspetto è la scalabilità, ovvero la capacità del sistema di mantenere invariate le prestazioni all'aumentare del numero di nodi sensori. I costi di produzione di un sistema sono in genere legati al costo dei singoli nodi sensori. Poiché una WSN prevede l'utilizzo di un elevato numero di sensori il costo associato ad ognuno di essi dovrebbe essere davvero basso al fine di realizzare una rete complessa. Il problema è però

che i nodi non sono semplici sensori, ma sono accompagnati da piccoli processori *onboard* e componenti che ne permettono la comunicazione wireless. A causa di questo i costi non sono poi così bassi e la realizzazione di un sistema complesso potrebbe comportare notevoli spese. Tuttavia esistono settori in cui i benefici forniti da sistemi come WSN superano di gran lunga l'importanza associata ai costi. Un altro importante aspetto da tenere in considerazione è quello legato ai consumi energetici. I sensori presenti in una WSN sono generalmente alimentati mediante una batteria e il mantenimento di una rete all'interno della quale la durata della batteria non è sufficientemente lunga per ogni sensore sarebbe poco pratico. Il consumo energetico di un nodo si può riassumere nella fase di *sensing* ovvero quella in cui vengono raccolti i dati, nella fase in cui tali dati vengono processati e infine quella in cui vengono comunicati al nodo centrale. Per ridurre al minimo il consumo energetico dei nodi sensori sono in corso svariate ricerche che puntano alla realizzazione di protocolli di tipo *power-aware* ovvero protocolli il cui obiettivo è quello di ottimizzare il consumo energetico.

Prima di addentrarci nella progettazione di un'applicazione distribuita, analizziamo i principali protocolli di comunicazione utilizzati in ambito di domotica ed *health monitoring*.

2.3 La comunicazione

Progettare un sistema capace di interfacciarsi con svariati dispositivi presenti già in commercio è certamente un lavoro arduo poiché risulta necessario che i dispositivi in questione e l'infrastruttura fissa parlino in qualche modo la stessa lingua. A tal proposito è di fondamentale

importanza lo studio dei protocolli di comunicazione utilizzabili per lo scambio di informazioni all'interno della rete domestica.

I protocolli di comunicazione possono dividersi in base alla tecnologia adottata per comunicare, che può essere wireless, wired o ibrida. Nel caso in esame, una comunicazione wired è ovviamente superflua: pensare ad un cablaggio di un gran numero di sensori in un'abitazione con una superficie molto ampia, non è il massimo. Tuttavia ciò non toglie che piccole parti del sistema potrebbero richiedere un cablaggio. Esamineremo più avanti nel dettaglio questo aspetto, per adesso possiamo limitarci a dire che si tratterà di un sistema ibrido in cui la grande maggioranza dei nodi adotterà un metodo di comunicazione wireless.

Analizziamo adesso vantaggi e svantaggi della comunicazione wireless.

Vantaggi

- **Mobilità:** è senza dubbio la caratteristica principalmente apprezzata. Grazie all'assenza di cavi, la posizione dei dispositivi interconnessi può variare senza alcun problema entro il raggio d'azione dei trasmettitori/ricevitori.
- **Flessibilità:** la struttura delle reti wireless favorisce l'integrazione semplice ed immediata di nuovi dispositivi. Questo rende l'utilizzo delle reti wireless estremamente flessibile e scalabile.
- **Costi ridotti:** non è necessario l'intervento di personale esperto per il collegamento fisico della rete, e i soli costi associati sono relativi ad oggetti fisici tra cui il nodo centrale, i nodi sensori ed eventuali attuatori.

Svantaggi

- **Sicurezza:** nelle comunicazioni cablate tutte le informazioni viaggiano all'interno di un cavo di rame. Ciò non succede per le comunicazioni wireless, nelle quali i dati sono contenuti in onde elettromagnetiche che si propagano attraverso l'etere, ragion per cui, nonostante varie tecniche di crittografia, sono maggiormente soggette ad intercettazioni ed eventuali decrittazioni.
- **Interferenze:** troppi segnali wireless, poiché operanti nella stessa banda, possono generare interferenze, peggiorando la qualità della comunicazione.
- **Copertura:** le reti wireless risentono della presenza di ostacoli quali muri o mobili. Ciò potrebbe essere un problema per applicazioni in cui è necessario che i segnali rimangano stabili.
- **Velocità:** la velocità di trasmissione nelle reti wireless è senza alcun dubbio minore rispetto a quelle cablate. Tuttavia risulta efficiente per applicazioni quali *smart home* in cui non si richiedono elevate velocità di trasmissione.

Esaminiamo adesso dei protocolli rappresentanti ormai uno standard nell'ambito della comunicazione wireless.

2.3.1 WiFi

Il protocollo *WiFi* è certamente uno dei protocolli wireless più diffuso. Lo standard più recente è l'802.11ac e risale al 2014. Opera in frequenze di 2.4

o 5GHz e riesce a coprire distanze di 50m per applicazioni indoor, arrivando anche a 100m per quelle outdoor. Lo standard precedente risale al 2009 e trasmette a velocità che si aggirano intorno ai 150-200Mbps, per arrivare ad un massimo di 300Mbps. Il più recente invece arriva a velocità 450Mbps sui 2.4GHz, e si estende fino a 1300Mbps sui 5GHz.

Il protocollo Wi-Fi è uno dei più conosciuti ed utilizzati in ambienti in cui è necessaria una connessione ad internet. Dal punto di vista dei consumi, non è di certo uno dei migliori protocolli, ed è questo il motivo per il quale non verrà trattato nel dettaglio. Come detto in precedenza infatti, l'obiettivo sarà integrare sensori e attuatori già presenti in commercio, i quali sono stati progettati per essere alimentati a batteria. Gli stessi produttori hanno quindi adottato protocolli di comunicazione mirati all'efficienza dal punto di vista energetico e di certo il Wi-Fi non costituisce una scelta ottimale.

2.3.2 Bluetooth

La tecnologia bluetooth fornisce un tipo di comunicazione a corto raggio e potrebbe funzionare molto bene per dispositivi all'interno di un'abitazione. Bluetooth opera ad una frequenza di 2,4GHz, utilizzando 79 canali che vanno da 2.402GHz a 2.480GHz. Supporta inoltre due tipi di comunicazione: quella sincrona (*SCO - Synchronous Connection Oriented*) e quella asincrona (*ACL - Asynchronous ConnectionLess*). La velocità di trasmissione dei dati in genere si aggira intorno a 1Mbps, ma può ridursi nel momento in cui vengono utilizzate comunicazioni SCO o ACL.

La comunicazione bluetooth standard avviene tra due dispositivi: uno master, l'altro, slave. Ogni nodo bluetooth può fungere sia da master, che da slave e possiede un indirizzo univoco a 48 bit.

La trattazione sul bluetooth sarà concentrata in particolar modo sulla sua versione a basso consumo (BLE - Bluetooth Low Energy) [10]. È infatti di fondamentale importanza che l'infrastruttura fissa e il dispositivo wearable (così come altri nodi della rete con interfaccia bluetooth e alimentazione a batteria) comunichino minimizzando il dispendio di energia, garantendo una durata della batteria quanto più ottimale possibile.

Il *BLE* si basa su una struttura a livelli come quella mostrata in Figura 9 - Sulla sinistra lo stack BLE, sulla destra lo stack bluetooth classico. In particolare, figura evidenzia le differenze dello stack bluetooth classico, con lo stack della sua versione a basso consumo energetico.

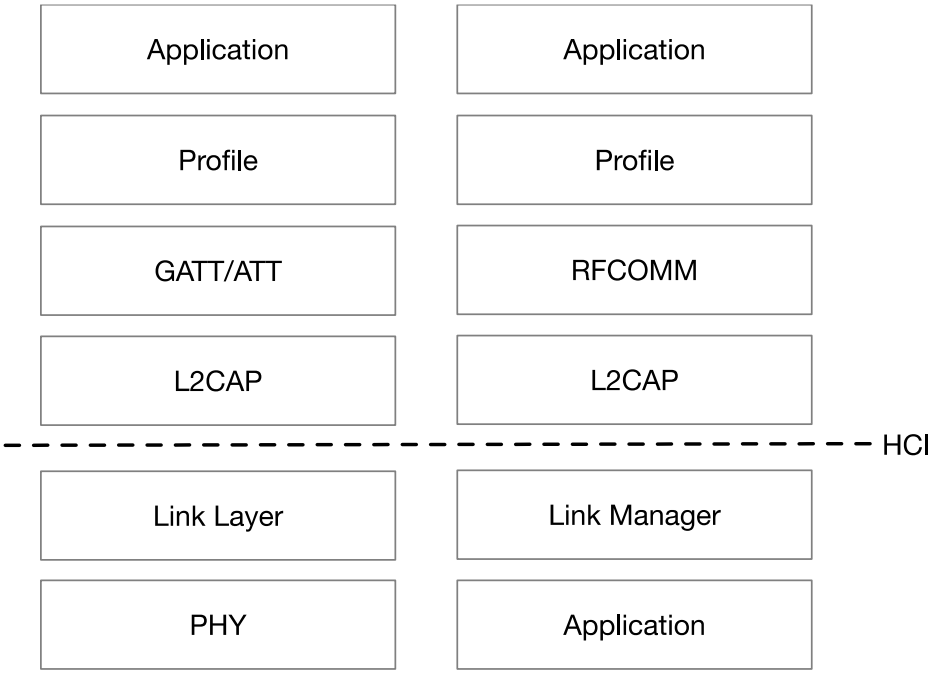

Figura 9 - Sulla sinistra lo stack BLE, sulla destra lo stack bluetooth classico

In questa tesi ci si concentrerà perlopiù sul funzionamento di *BLE* poiché si presta perfettamente per applicazioni come quella che descriveremo.

2.3.3 Bluetooth Low Energy (BLE)

Sebbene il sistema complessivo preveda una *WSN* composta da sensori di vario tipo, durante lo svolgimento di questa tesi ci si concentrerà maggiormente sull'interazione tra un nodo centrale e un dispositivo wearable di cui parleremo dettagliatamente più avanti. Poiché *BLE* è una tecnologia di comunicazione che riduce al minimo il consumo energetico è

molto utilizzato in ambito wearable, ed è adoperato dalla stragrande maggioranza di dispositivi di questo tipo quali smartwatch e smartband. Anche il dispositivo wearable con cui il nodo centrale comunicherà implementa *BLE* come tecnologia di comunicazione.

Il bluetooth a basso consumo nasce inizialmente come *Bluetooth Lite* a metà degli anni 2000 nei laboratori di ricerca *Nokia*. L'obiettivo era quello di definire una nuova versione dello standard bluetooth che si contrapponesse al suo predecessore dal punto di vista del consumo energetico. La Nokia mise su un'alleanza industriale con altre società, battezzata sotto il nome di *Wibree* al fine di lavorare a questo progetto, ma soltanto un anno dopo si resero conto che aveva più senso affidare questo incarico a *Bluetooth SIG*, l'organizzazione di standardizzazione responsabile dello sviluppo del bluetooth, nonché del rilascio di licenze e tecnologie ad esso collegate. La versione a basso consumo del bluetooth, fu introdotta a partire dalla sua versione 4.0, rilasciata nel 2010. Sin dal suo sviluppo, venne integrata in una moltitudine di dispositivi: dagli attuali smartwatch, alle comuni periferiche come PC, tablet e smartphone. I dispositivi bluetooth che adottano una tecnologia di tipo low energy, sono categorizzati in dispositivi bluetooth smart, e dispositivi bluetooth smart ready. I secondi, sono dispositivi che supportano la versione a basso consumo del protocollo e permettono che profili aggiuntivi possano essere integrati in seguito mediante l'installazione di mirate applicazioni e/o drivers. I primi invece sono in grado di comunicare esclusivamente con i dispositivi smart ready e non permettono di interfacciarsi con lo stack classico delle precedenti versioni di bluetooth (Figura 10).

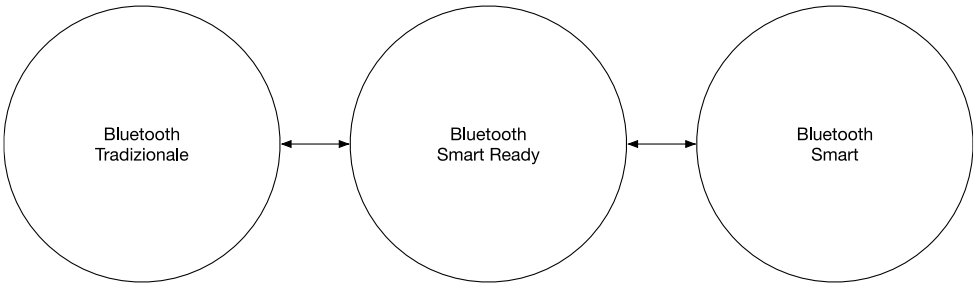

Figura 10 - Interfacciamento tra disposidivi bluetooth classici, smart ready e smart

Dal punto di vista tecnico esistono tre diversi tipi di dispositivi: Classic Bluetooth, Bluetooth Dual Mode e Bluetooth Single Mode. Il bluetooth classico abbraccia tutti quei dispositivi che necessitano di uno scambio costante di dati e di una connessione ad alto rendimento, basti pensare ad esempio a delle cuffie bluetooth, in cui i segnali audio costituiscono un'informazione consistente da trasmettere (centinaia di KB al secondo) con continuità nel tempo. I dispositivi bluetooth dual mode supportano invece la comunicazione sia con i dispositivi classici, che con quelli bluetooth smart ed è per questo che non ottengono alcun beneficio legato al consumo energetico, dovendo adottare delle tecnologie conformi al bluetooth classico. La categoria più interessante è rappresentata dai dispositivi bluetooth single mode, i quali supportano esclusivamente la tecnologia bluetooth low energy; non sono infatti in grado di comunicare con i dispositivi bluetooth classici, tuttavia sono estremamente ottimizzati per trarre il massimo vantaggio dalla comunicazione a basso consumo energetico. Si pensi ad esempio ad una fascia monitorante la frequenza cardiaca, la quale può rimanere connessa per diverse ore, trasmettendo pochi byte al secondo. Trasmettere qualche byte al secondo, dal punto di vista operativo significa che la fascia necessita di rimanere accesa per meno

di un millisecondo ogni secondo, ed è chiaro che questo abbassa drasticamente i consumi, garantendo una durata della batteria molto elevata.

Negli ultimi anni, sempre crescente è stato il numero di dispositivi BLE realizzati, in particolar modo in ambito sportivo, sebbene le sue applicazioni si possano estendere anche ad altri contesti come ad esempio quello sanitario. Un altro modo di pensare al bluetooth smart, è quello di abilitare il cosiddetto *Internet Of Things (IoT).* Anziché collegare i dispositivi *BLE* direttamente ad internet, essi si interfacciano prima a dispositivi Bluetooth Smart Ready come smartphone, tablet o PC e saranno questi ultimi ad occuparsi di far transitare i dati in rete. In questo modo non sarà necessaria alcuna infrastruttura aggiuntiva finalizzata all'inoltro in rete dei dati.

Uno dei layer più alti dello standard *BLE* è il protocollo *GATT*, che serve a raggruppare attributi in gruppi logicamente organizzati. Un attributo è un valore il quale ha associate tre proprietà:

- **UUID**: si tratta di un identificativo universalmente univoco, può essere a 16 o a 128 bit e serve ad identificare il tipo di attributo. Esistono UUID predefiniti dallo standard come *Dichiarazione di Servizio*, *Dichiarazione di Caratteristica* e altri.
- **Handle di attributo**: si tratta di un valore a 16 bit che identifica univocamente un attributo all'interno di un server. In questo modo un client può accedervi in lettura/scrittura senza ambiguità.
- **Autorizzazioni**: definiscono le regole mediante le quali si può interagire con un determinato attributo. Esse stabiliscono se un attributo può essere letto o scritto e quale tipo di autorizzazione è necessaria per eseguire tali operazioni. Le autorizzazioni vengono

applicate soltanto al valore dell'attributo, quindi non all'handle e neanche al campo autorizzazioni stesso.

Il valore di attributo può ovviamente essere qualsiasi cosa. Potrebbe ad esempio far riferimento ai battiti per minuto rilevati da un pulsiossimetro, allo stato di un LED o banalmente potrebbe essere costituito dalla semplice stringa *"Hello World"*.

Di seguito si riporta una tabella (Figura 11) che illustra un esempio pratico analizzando il profilo riguardante il monitoraggio della frequenza cardiaca. Come si può notare ogni riga rappresenta un attributo, ognuno dei quali ha un handle, un tipo, delle autorizzazioni e un valore.

Heart Rate Profile	Handle	Type of attribute (UUID)	Attribute permission	Attribute value
Service Declaration	0x000E	**Service Declaration** Standard UUID service 0x2800	Read Only, No Authentication, No Authorization	Heart Rate Service 0x180D
Characteristic Declaration	0x000F	**Characteristic Declaration** Standard UUID characteristic 0x2803	Read Only, No Authentication, No Authorization	Properties (Notify) Value Handle (0x0010) UUID for Heart Rate Measurement Characteristic (0x2A37)
Characteristic Value Declaration	0x0010	**Heart Rate Measurement Characteristic** UUID found in the Characteristic declaration value 0x2A37	Higher layer profile or implementation specific.	Beats Per Minute (E.g. "167")
Descriptor Declaration	0x0011	**Client Characteristic Configuration Descriptor (CCCD)** Standard UUID service 0x2800	Readable with no authentication or authorization. Writable with authentication and authorization defined by higher layer specification or is implementation specific.	Notification Enabled 0x000X
Characteristic Declaration	0x0012	**Characteristic Declaration** Standard UUID characteristic 0x2803	Read Only, No Authentication, No Authorization	Properties (READ), Value Handle (0x0011), UUID for Body Sensor Location (0x2A38)
Characteristic Value Declaration	0x0013	**Body Sensor Location** UUID found in the Characteristic declaration value 0x2A38	Higher layer profile or implementation specific	Sensor Location (8-bit integer) E.g. 3 equals "Finger"

Figura 11 - Esempio organizzazione di un servizio

Il protocollo *GATT* [11] si occupa di raggruppare gli attributi in gruppi logicamente organizzati. La tabella appena vista è un esempio di raggruppamento. Poiché tali gruppi sono organizzati proprio come la tabella precedente, la prima riga deve sempre contenere la dichiarazione di servizio, il cui *UUID* è *0x2800*. Il campo *handle* dei vari profili dipende dal numero di caratteristiche che ogni profilo espone. Il campo *value* contiene invece un *UUID* che indica il servizio che si vuole fornire. Nel caso del profilo per il monitoraggio del battito cardiaco, l'*UUID* contenuto nel campo *value* sarà proprio quello relativo al servizio di monitoraggio del battito cardiaco, cioè *0x180D*. Esiste una lista di *UUID* predefiniti, ma nulla toglie che per scopi ben precisi possano essere assegnati degli *UUID* custom. Dopo la dichiarazione di servizio, troviamo quasi sempre la dichiarazione di caratteristica, il cui *UUID* è *0x2803*. Nel campo *value* sono contenuti un *handle*, un *UUID* e un set di proprietà, che descrivono la riga della tabella riferita alla dichiarazione di valore caratteristico. L'*handle* indica la posizione all'interno della tabella della dichiarazione di valore caratteristico. L'*UUID* descrive il tipo di informazione o valore che possiamo aspettarci di trovare nella dichiarazione del valore caratteristico.

Nella dichiarazione del valore caratteristico troviamo finalmente il valore a cui siamo interessati, ad esempio un valore di temperatura o lo stato di un interruttore e l'*UUID* sarà lo stesso specificato nel campo *value* della dichiarazione di caratteristica.

La dichiarazione del valore caratteristico può essere seguita da tre entry differenti: ci potrebbe essere una nuova dichiarazione di caratteristica, una nuova dichiarazione di servizio oppure una dichiarazione di descrittore. Il descrittore è un attributo che fornisce informazioni aggiuntive sulla caratteristica cui fa riferimento.

2.3.4 Z-Wave e Zigbee

Z-Wave è un protocollo di comunicazione wireless a basso consumo energetico. È adoperato principalmente per applicazioni domotiche grazie al fatto che utilizza una topologia di rete di tipo mesh mediante la quale è possibile interconnettere dispositivi anche a distanze più elevate del solito. Ci sono infatti due principali componenti: i *controller* e gli *slave*. Nella rete è presente un controller principale che coordina il funzionamento di tutti i dispositivi. Gli slave sono invece i nodi come luci o interruttori e possono essere alimentati o a batteria.

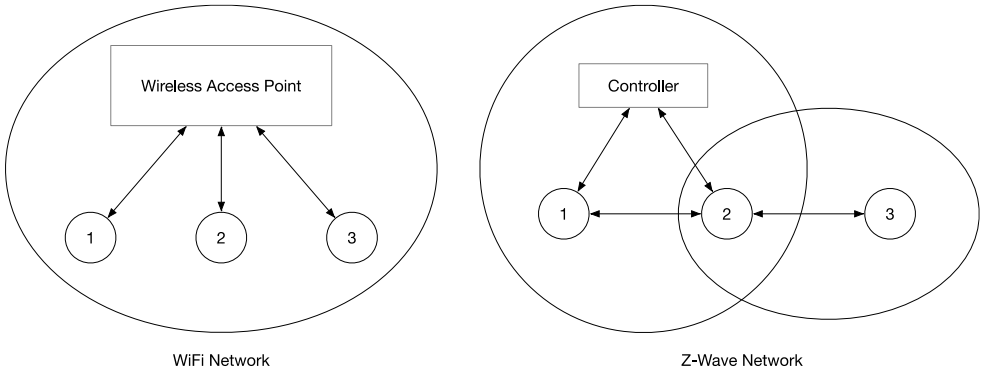

Figura 12 - Differenze tra una rete WiFi e Z-Wave

La Figura 12 evidenzia le differenze fondamentali che intercorrono tra una rete *WiFi* e una *Z-Wave*. In una rete Wi-Fi, tutti i nodi devono trovarsi alla portata del punto di accesso principale: il router. In una rete Z-Wave, invece, un generico nodo deve trovarsi semplicemente alla portata di un altro nodo. Un pacchetto può attraversare fino a 4 nodi, il che significa che questo aumenta notevolmente la distanza che può raggiungere a differenza delle classiche reti Wi-Fi. Affinché un nodo possa inoltrare un pacchetto,

deve essere alimentato. Un nodo non alimentato non sarà in grado di svolgere la funzione di inoltro. In genere, tutti i messaggi vengono inviati dal controller, ai nodi slave, i quali non sono abilitati ad avviare una comunicazione e possono soltanto rispondere. I nodi di inoltro invece possono inviare dei messaggi prestabiliti ai nodi associati. Per recapitare i dati dai vari nodi, il controller utilizza un meccanismo di polling mediante il quale richiede periodicamente i dati a ciascun nodo in modo da reagire tempestivamente ai vari eventi.

Quando un nodo *Z-Wave* non è incluso nella rete avrà un ID di rete e un ID di nodo entrambi pari a 0. Nel momento in cui viene incluso, l'ID di rete sarà quello del controller, e l'ID di nodo sarà un numero univoco compreso tra 2 e 232. Per l'ID di nodo vengono utilizzati 8 bit. L'ID di nodo del controller è pari ad 1. Quando un nodo viene eliminato, i suoi ID di nodo e di rete vengono impostati a 0.

Ogni nodo mantiene in memoria un elenco di nodi adiacenti grazie al quale si potranno creare le tabelle di routing.

Una rete Z-Wave richiede almeno un controller, chiamato controller primario. Possono tuttavia aggiungersi altri controller al fine di inviare comandi ai nodi. I controller possono essere di due tipi: statici e portatili. I primi sono alimentati e rimangono fissi nel punto in cui vengono piazzati. Quelli portatili invece sono alimentati a batteria e non possono instradare messaggi. Tutti i controller posseggono una copia della tabella di routing.

Il protocollo *Zigbee*, funziona in modo molto simile a *Z-Wave*: anch'esso sfrutta una topologia di rete di tipo mesh. Poiché la logica che sta dietro entrambi i protocolli è grossomodo la stessa, elenchiamo le maggiori differenze tra i due.

Zigbee opera a frequenze di 915MHz e 2.4GHz, la stessa frequenza a cui opera il *WiFi*. Una cosa di questo tipo potrebbe causare interferenze in ambienti in cui sono presenti un numero elevato di dispositivi *WiFi* e *Zigbee*. Sotto questo punto di vista, *Z-Wave* non soffre di questa lacuna, poiché opera a frequenze di 800-900MHz. Dal punto di vista della velocità di trasmissione, *Zigbee* ha prestazioni migliori di *Z-Wave*: la sua velocità di trasmissione infatti risulta essere tra i 40 e i 250 Kbps, contro i 10-100 Kbps di *Z-Wave*. Le applicazioni per le quali vengono utilizzati tali protocolli non richiedono tuttavia elevate velocità di trasmissione, infatti entrambi i protocolli risultano estremamente lenti paragonati al *WiFi*, ma ciò non è rilevante ai fini della realizzazione di sistemi domotici, per i quali le uniche informazioni transitanti possono riguardare istruzioni da fornire ad un dispositivo per eseguire una determinata azione o letture da sensori. Un'altra differenza tra questi protocolli risiede nel numero di nodi collegabili alla rete: mentre *Z-Wave* fornisce un limite di nodi collegabili pari a 232, *Zigbee* permette di collegarne 65000. Entrambi i protocolli utilizzano AES-128 per mantenere la casa al sicuro, in particolar modo quando si tratta di gestire nodi sensibili come serrature intelligenti o sensori di movimento di sistemi d'allarme. Così come *Z-Wave*, *Zigbee* fornisce una moltitudine di dispositivi già presenti in commercio come luci, elettrodomestici, termostati o lucchetti smart, ma nella maggior parte dei casi, i dispositivi che supportano *Z-Wave*, supportano anche *Zigbee*.

2.3.5 Altri protocolli

Dopo l'introduzione di BLE, sono state proposte altre soluzioni wireless a basso consumo: è il caso di EnOcean, il quale permette comunicazioni fino

a 30 metri all'interno degli edifici e 300 all'esterno, oppure 6LoWPAN, una versione a basso consumo dell'IPv6, o ancora il protocollo LoRa, utilizzato però in contesti urbani nella realizzazione delle *"Smart-Cities"*.

2.4 MQTT

Data una topologia di rete sulla quale viene costruita una WSN è importante definire il modo in cui i dispositivi che ne fanno parte devono comunicare. Anche se in questa tesi non si scenderà in dettaglio sull'argomento è sicuramente interessante accennare MQTT poiché definisce delle regole mediante le quali una rete di sensori può comunicare. MQTT (Message Queuing Telemetry Transport) è un protocollo a livello applicativo semplice e versatile e può collocarsi quindi alla cima dello stack protocollare TCP/IP. È stato ideato nel 1999 al fine di ottenere un protocollo che potesse ridurre al minimo il consumo energetico ed ottimizzare la comunicazione tra dispositivi che prevedono un'alimentazione a batteria. MQTT sfrutta un meccanismo di pubblicazione e sottoscrizione mediante un componente chiamato broker, in grado di gestire migliaia di client contemporaneamente. I client devono essere in grado di comunicare tra loro, e raggiungono tale obiettivo utilizzando i cosiddetti topics. Un topic non è altro che una sorta di chat, contraddistinta da un nome ben preciso definito dal programmatore, sulla quale i publisher pubblicano dati e dalla quale i subscriber leggono, sottoscrivendosi al topic a cui sono interessati. Il message broker è il componente che si occupa di filtrare e distribuire le comunicazioni tra publishers e subscribers.

MQTT è un protocollo molto interessante poiché permette l'installazione di nuovi componenti domotici lasciando inalterata la rete attuale. L'installazione ad esempio di una nuova coppia sensore-attuatore, dal punto di vista applicativo, consisterebbe semplicemente nel far agire il sensore da publisher, l'attuatore da subscriber e definire i comportamenti azione-reazione che si vogliono implementare.

3 Infrastruttura hardware e software

Questa sezione si concentra finalmente sulla fase sperimentale del progetto in esame. Avevamo precedentemente visto l'idea di come il sistema completo dovrebbe funzionare. Durante questo progetto di tesi ci si occuperà di sviluppare la parte del sistema relativa alla comunicazione tra un dispositivo wearable e un dispositivo fisso. Nello specifico ci si dedicherà allo sviluppo di un'applicazione distribuita per il rilevamento delle cadute e del modo in cui gli eventi vengono comunicati ad uno smartphone remoto posseduto dal *caregiver*. Al fine di testare tale applicazione sarà necessario operare mediante un dataset contenente i dati realtivi a vari tipi di cadute. Verrà sviluppata quindi una funzionalità di virtualizzazione di cui si parlerà nel dettaglio nel Capitolo 4 che permetterà di testare il corretto funzionamento del sistema.

3.1 Dispositivi adoperati

Come detto precedentemente gli attori principali della parte del progetto che verrà sviluppato sono due: un dispositivo wearable ed un dispositivo fisso. Il primo, rappresentato da una piccola scheda di sviluppo con a bordo svariati sensori, è di proprietà della ST Microelectronics e prende il nome di STEVAL-STLKT01V1 o SensorTile [12]. Si parlerà più nello specifico della SensorTile nella tesi a questa complementare. Il dispositivo fisso, su cui invece ci si concentrerà in questo contesto, è rappresentato da un RaspberryPi che analizzeremo di seguito in dettaglio.

3.1.1 RaspberryPi

Il dispositivo fisso scelto per lo sviluppo di questo progetto è il RaspberryPi 3 B+ (Figura 13) [13]. Si tratta di una delle schede di sviluppo più popolari in circolazione. Presenta un processore quad core a 64 bit operante ad una frequenza di 1.4GHz, 1GB di memoria RAM e moduli integrati quali *WiFi* e *BLE*. La possibilità di utilizzare una scheda micro-SD per contenere dati e sistema operativo rende questo dispositivo estremamente capiente considerando le sue dimensioni ridotte. Ai fini del sistema proposto si presta perfettamente poiché è a tutti gli effetti un computer che riducendo gli spazi al minimo, può essere posizionato strategicamente in qualsiasi punto della propria abitazione. La presenza di un sistema operativo permette di far girare applicazioni scritte nei più moderni linguaggi di programmazione ad alto livello. Vedremo infatti più avanti il modo in cui il dispositivo fisso sarà capace di far girare un programma server sfruttando moderne tecnologie come Node.js.

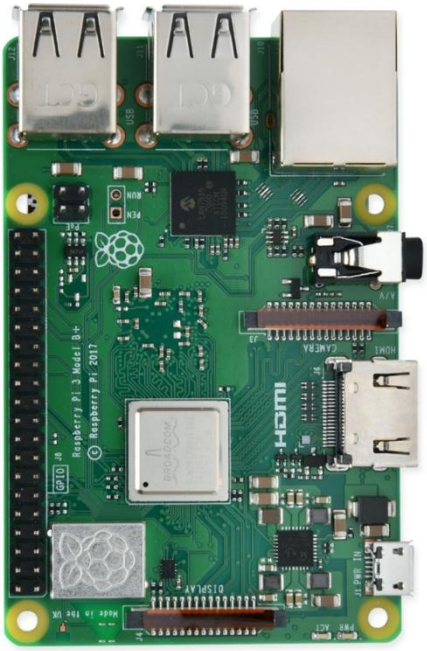

Figura 13 - RaspberryPi 3 B+

3.2 Interfacciamento col dispositivo wearable

Nel capitolo precedente è stato analizzato il funzionamento della tecnologia *BLE*. Prima di procedere con lo sviluppo di un'applicazione distribuita che coinvolga il nodo fisso e il nodo wearable è necessario comprendere a pieno dal punto di vista pratico come i due dispositivi possano comunicare, mettendo in campo le nozioni apprese sul funzionamento del protocollo *GATT*. Dopo aver effettuato l'accesso al Raspberry mediante protocollo *SSH* saremo in grado di accedere a tutte le funzionalità che la shell di linux mette a disposizione. Ogni dispositivo fisico è identificato mediante un codice univoco chiamato indirizzo *MAC*. La connessione al dispositivo wearable deve avvenire conoscendone tale indirizzo. L'interfaccia *hcitool* viene utilizzata per configurare le connessioni bluetooth ed inviare comandi speciali ai dispositivi bluetooth nelle vicinanze. Il nome *hcitool* deriva dal fatto che per far comunicare il livello collegamento con i livelli più alti dello stack *BLE* (Figura 14) viene utilizzata una porta *hci (Host Controller Interface)*.

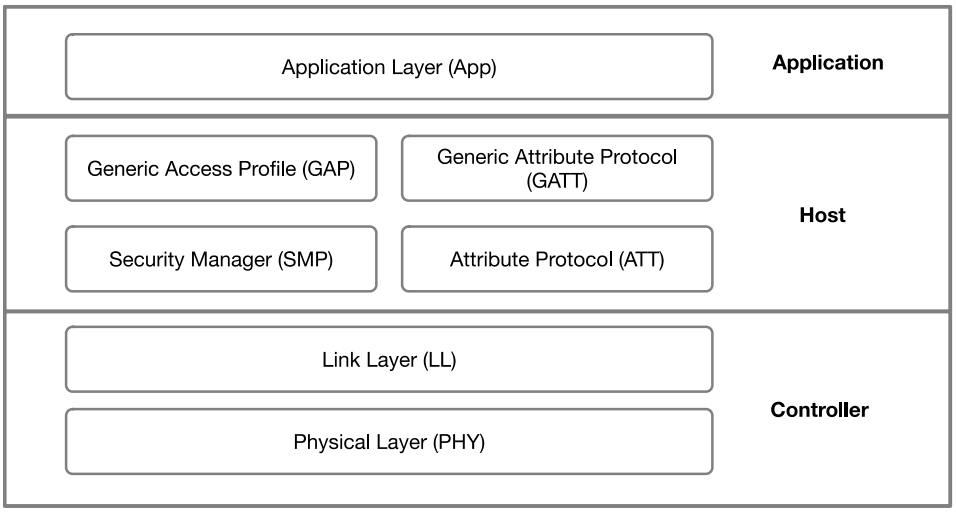

Figura 14 - Stack BLE

La comunicazione tra host e controller avviene proprio mediante *hci*. In generale host e controller risiedono in locazioni differenti. Mentre l'host gira direttamente nella CPU, il controller è parte dell'hardware del chip bluetooth. Nel caso di uno smartphone ad esempio il controller si trova nel chipset bluetooth e l'host fornisce un'interfaccia mediante la quale gli sviluppatori possono accedere alle funzionalità bluetooth in modo astratto indipendentemente dal chip con cui hanno a che fare. Le specifiche bluetooth definiscono *hci* come un insieme di comandi ed eventi per l'host ed è proprio tramite tali comandi che è possibile interfacciarsi ai dispositivi.

Nel nostro caso specifico *hcitool* verrà utilizzato con il comando *lescan* (Figura 15), che permette la scansione dei dispositivi *BLE* nelle vicinanze. Come risultato si ottiene una lista in cui ogni elemento è composto dall'indirizzo *MAC* del dispositivo trovato e dal nome simbolico ad esso associato.

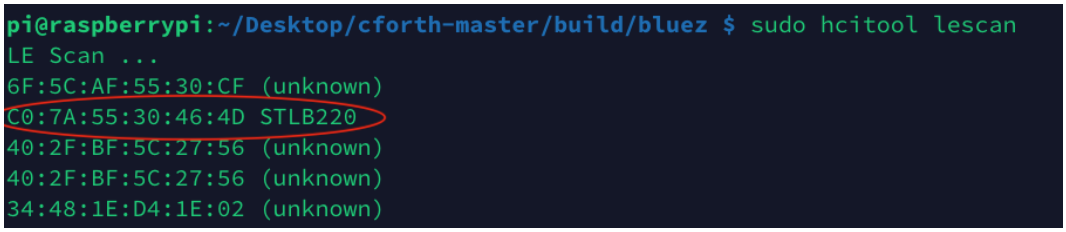

Figura 15 - Scansione dispositivi BLE da riga di comando

Come è possibile osservare, tra i dispositivi scansionati è presente quello a cui siamo interessati. Il nome simbolico associato alla SensorTile è *STLB220* e il suo indirizzo *MAC, C0:7A:55:30:46:4D.*

Un'altra interfaccia che verrà utilizzata è *gatttool*, che serve ad accedere alle caratteristiche esposte da dispositivi *BLE* e fonrnisce la possibilità di scrivere, leggere o sottoscriversi ad esse.

Si riporta di seguito una schermata (Figura 16) in cui si è effettuata una sessione di connessione al dispositivo wearable.

Figura 16 - Esplorazione di servizi e caratteristiche del dispositivo wearable da riga di comando

Dopo aver effettuato la connessione al dispositivo wearable sono stati letti i servizi esposti da questo mediante il comando *primary*. Come è possibile notare i servizi sono cinque ed ognuno di essi possiede un UUID univoco, un handle di inizio ed uno di fine. Ogni servizio, come avevamo già avuto modo di vedere, al suo interno contiene una o più caratteristiche. Per elencare le caratteristiche del dispositivo wearable viene utilizzato il comando *characteristics*. Osservando i valori degli handle sarà possibile capire quali caratteristiche appartengono a quali servizi. A tal proposito è stato realizzato uno schema riassuntivo (Figura 17). Per questioni di leggibilità sono riportati esclusivamente gli handle.

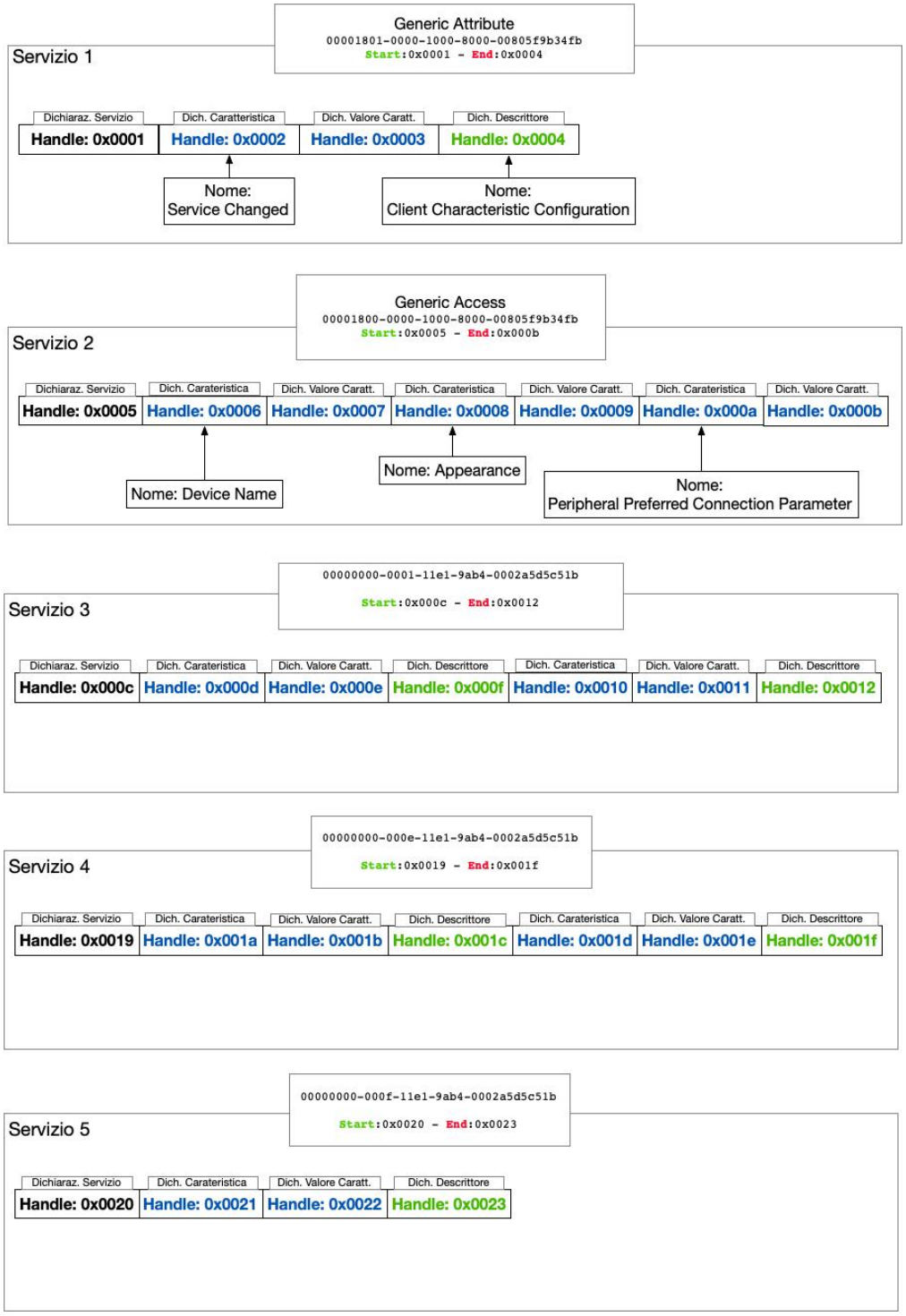

Figura 17 - Schema riassuntivo dei servizi e delle caratteristiche del dispositivo wearable

A questo punto è necessario saper leggere da una caratteristica, scrivere su una caratteristica e sottoscriversi alle notifiche di una caratteristica che presenta un permesso di tipo *notify*. Per farlo ci si avvale sempre dell'interfaccia software *gatttool*. Per testare il corretto funzionamento della lettura digitiamo il comando *char-read-hnd 0x0007*, col quale si andrà a leggere il valore della caratteristica con handle *0x0007*, ovvero la caratteristica contenente il nome simbolico del dispositivo wearable. Si noti che i caratteri sono codificati in esadecimale (Figura 18).

Figura 18 - Lettura del valore di una caratteristica mediante hanlde da riga di comando

Nella SensorTile al momento è presente una routine che scrive costantemente la stringa *"Hello World"* nella caratteristica con handle *0x001c*. Al fine di ricevere le notifiche ogni qualvolta la caratteristica viene scritta è necessario gestire gli eventi di notifica. Scrivendo nella caratteristica con handle *0x001c* il valore esadecimale *0x0100* vengono abilitate le notifiche. Scrivendo invece *0x0000* vengono disabilitate (Figura 19).

```
[C0:7A:55:30:46:4D][LE]> char-write-cmd 1c 0100
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
Notification handle = 0x001b value: 48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0d 0a
[C0:7A:55:30:46:4D][LE]> char-write-cmd 1c 0000
```

Figura 19 - Abilitazione e disabilitazione di notifiche mediante handle della caratteristica utilizzando la riga di comando

Abbiamo visto come è stato possibile interagire col dispositivo wearable da linea di comando. Purtroppo ai fini della realizzazione di un'applicazione distribuita ciò non basta. È necessaria una piattaforma che permetta di scrivere gli algoritmi che il dispositivo fisso deve eseguire durante l'interazione col dispositivo wearable. Poiché oltre l'interazione con i vari nodi del sistema il dispositivo deve fungere anche da server, per scrivere l'applicazione distribuita è stato scelto Node.js, un linguaggio lato server molto versatile e per il quale sono presenti un gran numero di librerie

di terze parti. Descriveremo di seguito Node.js e parleremo delle librerie adoperate per costruire un'applicazione distribuita per il rilevamento di eventi.

3.3 Node.js

Durante i primi anni 2000 cominciò a spopolare JavaScript come linguaggio di programmazione lato client per lo sviluppo di applicazioni web. Col passare degli anni il linguaggio venne migliorato sempre più fino a quando nel 2008 la Google integrò in Chrome il motore JavaScript V8, avente delle performance notevolmente migliorate tanto da mettere in difficoltà le case produttrici di web browser avversari. Node.js è una piattaforma costruita a partire dal motore JavaScript V8. Se V8 è nato inizialmente allo scopo di ottenere un supporto migliore per JavaScript all'interno del browser, con Node è possibile adesso scrivere robuste applicazioni lato server, cosa che era miseramente fallita con i motori JavaScript precedenti. Node, quindi, non è altro che il motore JavaScript V8 estratto dal browser e reso indipendente da quest'ultimo. Una grande potenzialità di Node.js è quella di operare in modo asincrono. Esso infatti fornisce un meccanismo per realizzare applicazioni server scalabili senza impiegare un'eccessiva potenza di calcolo, segue infatti un sistema di tipo *event-driven*, ovvero viene attivato soltanto nel momento in cui arrivano delle richieste. In quel caso viene allocata una piccola quantità di memoria utilizzata per capire quale procedura eseguire in base all'evento ricevuto. La procedura eseguita prende in generale il nome di *callback*. Grazie alla presenza delle *callback* Node.js riesce a gestire le operazioni I/O in modo non

bloccante ed è questo uno dei principali punti forza del linguaggio in questione. Un sistema di questo tipo supera quindi tutte le difficoltà legate alla programmazione concorrente sia in termini di complessità di codifica, sia in termini di risorse allocate per ogni thread. Gli usuali linguaggi di programmazione lato server come Java o PHP hanno una gestione dell'I/O bloccante. Ciò significa che ogni volta che arriva una richiesta da soddisfare viene creato un thread che verrà distrutto nel momento in cui la richiesta è portata a termine. Si immagini il caso in cui tante richieste arrivino contemporaneamente. In questo caso i sistemi non possono definirsi scalabili poiché più richieste arrivano contemporaneamente, più thread vengono allocati e più le risorse del sistema saturano. Node invece funziona diversamente. Si immagini di avere un unico grande thread il quale crea una *callback* ogni volta che riceve una richiesta. In questo modo nessun processo viene messo in attesa e le prestazioni del sistema vengono svincolate dal numero di richieste ricevute. Un sistema di questo tipo scala molto bene.

Dal punto di vista delle performance c'è da dire che il garbage collector di JavaScript V8 posiziona gli oggetti in memoria in modo casuale. A causa di questo, a fronte di operazioni di output come il trasferimento dei dati attraverso un socket il tempo di trasferimento aumenta all'aumentare della dimensione dei dati da trasferire. Per limitare questo problema una possibile soluzione è quella di utilizzare l'oggetto *Buffer* tramite il quale è possibile lavorare direttamente con sequenze di byte. È chiaro che se le dimensioni dei dati da trasferire sono eccessive il problema non viene del tutto risolto. Poiché l'applicazione che verrà realizzata vedrà operazioni di output di pochi byte, il problema sui tempi di trasferimento non si pone.

Per scrivere un'applicazione distribuita per il rilevamento di eventi è necessario appoggiarsi a moduli esterni scaricabili gratuitamente mediante uno dei *package manager* più estesi al mondo: *npm (Node Package Manager).* Tramite *npm* è possibile installare tutte le librerie aggiuntive di cui l'applicazione ha bisogno. Nel caso specifico dell'applicazione in esame è presente la necessità di mettere un server web in ascolto di richieste, utilizzare i socket come canale di comunicazione mediante la libreria *SocketIO* e avere delle funzioni di alto livello che permettano di interagire con dispositivi *BLE* utilizzando *Noble*, un modulo per la gestione di dispositivi bluetooth a basso consumo energetico.

3.3.1 SocketIO

SocketIO è una libreria JavaScript che permette la realizzazione di applicazioni web in tempo reale [14]. Fornisce la possibilità di avere una comunicazione bidirezionale tra client e server. SocketIO è composta da una parte client che solitamente gira in un browser web, e una parte server costruita su una libreria per Node.js. Caratteristica fondamentale di questa libreria è il fatto che è orientata agli eventi e funziona mediante l'utilizzo di funzioni di *callback*. Supponiamo di avere un server al quale è collegato un LED ed un client che da remoto vuole accendere il LED. Il server aprirà quindi un socket attraverso il quale accoglierà le richieste dei client. La libreria SocketIO prevede che ci siano dei topic sui quali pubblicare i dati. Per creare un topic l'unica cosa da fare è definirne un comportamento tramite il metodo *on* che prende in ingresso due parametri: il primo è il nome del topic attraverso il quale il dato viene ricevuto, il secondo è la funzione da eseguire alla ricezione del dato. Nel momento in cui un client

pubblica su un determinato topic, viene quindi eseguita la funzione di *callback* dal server in relazione al topic su cui sono stati ricevuti i dati. Supponiamo che il server abbia creato un topic chiamato *"led"*. Quando su questo topic viene ricevuto *true,* il LED deve accendersi, e deve spegnersi nel caso in cui venga ricevuto *false*. Per far questo il server deve definire una funzione di *callback* chiamata ogni qualvolta qualcuno pubblica sul topic *"led"*. Nel momento in cui il server definisce la *callback* viene effettivamente istanziato il topic *"led"* e qualunque client connesso al server potrà pubblicare dati su quel topic attraverso il socket sul quale il server è in ascolto.

```
socket.on("led",data =>{
    if(data==true){
        // Accendi il led
    }
    else{
        // Spegni il led
    }
})
```

Una volta definita la *callback* il server non farà nulla finché qualcosa non viene pubblicato sul topic *"led"*. Ciò sarà compito del client il quale si occuperà di effettuare una *emit*.

```
socket.emit("led",true)
```

Così come il client pubblica sui topic, può farlo anche il server. In questo caso è il client a dover definire una funzione di *callback* per gestire l'evento che il server genera. Questo meccanismo ad alto livello fornisce un canale di comunicazione bidirezionale tra client e server.

3.3.2 Noble

Noble è una libreria per Node.js che permette di interfacciarsi con dispositivi bluetooth a basso consumo energetico mediante delle API di alto livello semplici ed intuitive [15]. Nel codice che segue ad esempio è riportata la funzione che permette di connettersi al dispositivo wearable. L'indirizzo del dispositivo *BLE* è contenuto all'interno della variabile *PERIPHERAL_ID,* mentre *allServices* e *allCharacteristics* sono delle variabili in cui andremo a memorizzare una lista di oggetti di tipo servizio e caratteristica.

```
const connection = () => {
  noble.on('stateChange', function(state) {
    if (state === 'poweredOn') noble.startScanning()
    else noble.stopScanning()
  });
  noble.on('discover', peripheral => {
    if(peripheral.id == PERIPHERAL_ID){
      noble.stopScanning();
      peripheral.on('connect', () => {
        peripheral.discoverAllServicesAndCharacteristics((error, services, characteristics) => {
          allServices = services
          allCharacteristics = characteristics
          let servUuids = []
          let charUuids = []
          for(let i=0 ; i<allServices.length;i++){
            servUuids.push(allServices[i].uuid)
          }
          for(let i=0 ; i<allCharacteristics.length;i++){
            charUuids.push(allCharacteristics[i].uuid)
          }
        })
      })
      peripheral.connect()
    }
  })
}
```

Come è possibile notare la libreria funziona sfruttando funzioni di *callback* che vengono scatenate nel momento in cui un particolare evento si verifica.

Una volta eseguita la funzione che stiamo esaminando, il nodo fisso sarà connesso con il dispositivo wearable e sarà possibile interagirvi mediante altre funzioni che consentono di interfacciarsi con il protocolo *GATT*, in particolare funzioni per leggere, scrivere e sottoscriversi alle notifiche di una determinata caratteristica. Si riportano di seguito le firme di tali metodi.

Il metodo *write*, invocato su una caratteristica, permette di scrivere su quest'ultima. L'oggetto *data* deve essere un oggetto di tipo *Buffer*. In Node.js i buffer sono utilizzati per la rappresentazione di dati come sequenze di byte. Le caratteristiche sulle quali andremo a scrivere i valori saranno in particolare capienti 20 Byte ed è importante non superare tale limite. Il parametro *withoutResponse* è un valore booleano che, se impostato a *true*, permette di non aspettare la conferma di ricezione del messaggio prima di inviare il successivo. È chiaro che impostare tale parametro a *false* implica un rallentamento dell'intero sistema nel momento in cui è necessario inviare tanti dati in breve periodo di tempo. Il terzo parametro è opzionale ed è il riferimento alla funzione di *callback* chiamata nel momento in cui la scrittura termina. Tramite il parametro *error* sarà possibile gestire situazioni impreviste.

```
characteristic.write(data, withoutResponse[, callback(error)]);
```

Il metodo *read*, invocato su una caratteristica permette di leggerne il valore contenuto. L'unico parametro in ingresso è una funzione di *callback* che verrà chiamata a lettura completata. Mediante i parametri *error* e *data* potremo rispettivamente gestire situazioni impreviste e interagire con i dati letti.

```
characteristic.read([callback(error, data)]);
```

Il metodo *subscribe*, invocato su una caratteristica permette di abilitare le notifiche di quest'ultima. Opzionalmente è possibile anche in questo caso gestire tramite una *callback* situazioni di errore. È importante sottolineare la differenza tra una lettura e una sottoscrizione poiché apparentemente sono simili. Un'operazione di lettura può essere effettuata in qualsiasi momento per leggere il valore all'interno di una caratteristica. La notifica invece viene scatenata nel momento in cui il valore della caratteristica viene modificato. In quel momento di fatto è come se si stesse effettuando una lettura, con la sola differenza che avviene non appena dentro la caratteristica c'è qualcosa di nuovo. In modo complementare è presente il metodo *unsubscribe*.

```
characteristic.subscribe([callback(error)]);
```

Una volta effettuata una sottoscrizione ad una caratteristica, è necessario assegnare un comportamento da assumere alla ricezione di una notifica da parte di quella caratteristica. Per far ciò viene utilizzato il metodo *on* al quale come primo parametro viene passata la stringa *"data"*. Il secondo parametro è invece una *callback* che verrà chiamata col dato effettivo come parametro. Niente da dire invece per il parametro opzionale *isNotification* utilizzato per scopi di compatibilità e che è ormai deprecato.

```
  characteristic.on('data', callback(data, isNotification));
```

Noble contiene tantissimi altri metodi e funzionalità che non verranno approfondite nell'ambito della tesi in esame. Per ulteriori approfondimenti è possibile consultare la repository ufficiale su *GitHub*.

3.4 Applicazione distribuita per il rilevamento delle cadute

Alla luce di quanto visto, è adesso possibile costruire un'applicazione distribuita finalizzata al rilevamento di eventi. In particolare l'interesse è ricaduto su un'applicazione distribuita per il rilevamento delle cadute mediante i dati rilevati da un sensore accelerometro posto all'interno del dispositivo wearable che sarà indossato dal paziente. Del modo in cui le cadute vengono rilevate se ne parlerà nella tesi complementare a quella in questione. In questa tesi invece è possibile più in generale parlare di applicazione distribuita per il rilevamento di eventi poiché il funzionamento è identico per qualsiasi tipo di evento di allarme.

Nel capitolo precedente è stata analizzata la struttura interna del dispositivo wearable, in particolare si è vista nel dettaglio l'organizzazione delle caratteristiche *BLE* e dei servizi che le contengono. Per scopi sperimentali è stata aggiunta al dispositivo wearable una nuova caratteristica che verrà utilizzata per contenere eventi rilevati dal dispositivo indossabile. Non ci concentreremo su come questa caratteristica è stata creata poiché lato dispositivo fisso l'unica cosa a cui siamo interessati è l'*UUID* e che tale caratteristica esponga tra le proprietà quella di *notify*.

Sul dispositivo wearable girerà una routine che si occuperà di monitorare i dati forniti dall'accelerometro al fine di rilevare eventuali cadute. Se dovesse rilevare una caduta scriverà sulla caratteristica *BLE* di cui si parlava pocanzi un messaggio. Tale messaggio avrà una struttura molto semplice: immaginiamo una semplice informazione testuale che non superi 20 byte, la massima capienza della caratteristica. Poiché è opportuno classificare i messaggi ricevuti almeno in basse alla criticità, potrebbe essere utile utilizzare il primo byte per informazioni sul tipo di messaggio. Il resto dei byte potrebbe essere dedicato al messaggio effettivo che dovrà essere inviato, ad esempio *"Fall detected"*.

Poiché il dispositivo fisso sarà sottoscritto alle notifiche della caratteristica verrà scatenata una *callback* che si occuperà di gestire l'evento generato. Il dispositivo fisso ha il compito di prendere questo messaggio, aggiungere un timestamp ed inviarlo tramite un socket al dispositivo client remoto in possesso del *caregiver*. Tale dispositivo, nel più semplice dei casi, potrebbe essere uno smartphone il quale riceve una notifica a seguito della quale il *caregiver* deciderà come operare al di fuori dell'applicazione distribuita. Potrebbero implementarsi sistemi più sofisticati all'interno dei quali piuttosto che notificare l'accaduto ad uno smartphone, inviino un messaggio ai soccorsi tramite un modulo *GSM*. È chiaro che è possibile utilizzare un approccio ibrido tramite il quale il dispositivo fisso analizza il messagio ricevuto dal dispositivo wearable e a seconda del tipo di messaggio decide se notificare il *caregiver* o contattare direttamente del personale di soccorso specializzato. Se l'evento di allarme è associato ad una caduta la cosa migliore potrebbe essere contattare il *caregiver* il quale si occuperà di verificarne la gravità. Se l'evento è invece associato ad un aumento insolito della frequenza cardiaca contattare prima il servizio

sanitario potrebbe essere decisivo. Se abbiamo a che fare invece con un evento associato ad un incendio la scelta giusta sarebbe avvisare tempestivamente i vigili del fuoco. Poiché il *caregiver* in generale è rappresentato da chi si occupa della persona assistita la cosa migliore potrebbe essere quella di avvisarlo comunque indipendentemente dal tipo di situazione.

Di seguito si riporta un esempio di codice in cui il server rappresentato dal nodo fisso si sottoscrive alla caratteristica sulla quale un ipotetico evento verrà scritto. Viene quindi definita una funzione da eseguire nel momento in cui un messaggio di allarme viene generato.

Quando il nodo wearable rileva una situazione di allarme, si occuperà di scrivere nella carattestica il messaggio associato. Così facendo scatterà una notifica che verrà rilevata dal nodo fisso e conterrà ad esempio il messaggio testuale in riferimento al tipo di allarme. Il nodo fisso genererà un timestamp ed emetterà sul topic *alarm_event* l'array composto dal messaggio di allarme e dal timestamp.

```
eventCharacteristic.subscribe()

eventCharacteristic.on("data" , alarmMessage => {
    let timestamp = new Date().getTime()
    io.emit("alarm_event" , [alarmMessage, timestamp] )
})
```

Al topic sul quale viene inviato l'allarme sarà sottoscritto il programma che gira ad esempio sullo smartphone del *caregiver*, collegato costantemente in background con il server che gira nel nodo fisso. Possiamo ad esempio supporre che venga istanziata una notifica push nel momento in cui il *caregiver* riceve il messaggio di allarme. In termini di sicurezza sarebbe utile

fare in modo che ci si assicuri che il *caregiver* o chi di dovere abbia ricevuto il messaggio. Per far ciò basterebbe confermare la corretta ricezione dell'allarme dall'applicazione mobile, la quale sempre tramite socket invierà un evento di avvenuta ricezione al server.

4 Sistemi di verifica mediante virtualizzazione

Spesse volte scrivere un'applicazione distribuita potrebbe non essere semplice, ad esempio nei casi i cui è necessario effettuare dei test che l'ambiente di lavoro non consente di mettere in pratica. Si immagini di dover scrivere un'applicazione per una *WSN* che prevede il coinvolgimento di sensori che rilevino situazioni di allarme come ad esempio sensori antincendio. È chiaro che sarebbe impossibile far scaturire un incendio ogni qualvolta l'applicazione deve essere testata. L'esempio dell'applicazione distribuita per il rilevamento delle cadute vista al capitolo precedente può essere considerato altrettanto valido.

4.1 Virtualizzazione di un accelerometro

Il contesto universitario in cui si lavorava durante lo sviluppo del progetto non rendeva possibile effettuare dei test specifici per raccogliere i dati dell'accelerometro durante una caduta. Un altro problema che rende il test con sensori reali ancora più arduo è legato all'impossibilità di testare il corretto funzionamento di un algoritmo su dati ben specifici. Un algoritmo per il rilevamento delle cadute ad esempio potrebbe comportarsi in modo particolare con una determinata caduta, e se si volesse verificare un diverso comportamento dell'algoritmo su quella stessa caduta, non sarebbe possibile utilizzando esclusivamente sensori reali. Per ovviare a questo problema è necessario fornire la possibilità di testare diversi algoritmi sugli

stessi dati in modo tale da verificarne la robustezza e ridurre al minimo il rilevamento di falsi positivi. A tale scopo è d'obbligo munirsi di dati ottenuti tramite terze parti raccolti in contesti di lavoro opportuni e con mezzi adeguati. Per la realizzazione dell'algoritmo di rilevamento delle cadute, ad esempio, è stato utilizzato un dataset chiamato *Sisfall* [16] contenente 19 tipi di ADL (Activity Daily Living, ovvero le attività giornaliere) e 15 tipi di cadute differenti. I dati sono stati raccolti tramite un dispositivo allacciato alla vita del soggetto e hanno coinvolto 38 persone, di cui 23 tra i 19 e i 30 anni, e 15 tra 60 e 75. Si parlerà del dataset nella tesi a questa complementare. Per la trattazione seguente ci servirà soltanto essere a conoscenza del fatto che ogni test di caduta ha una durata di 15 secondi e i dati relativi ad ogni test del dataset sono campionati ad una frequenza di 200Hz, quindi per ogni caduta avremo un totale di 3000 campioni. Il contenuto del file di testo contenente una caduta è organizzato per righe. In ogni riga sono presenti i valori dei tre assi dell'accelerometro separati da virgola.

Il capitolo corrente di questa tesi si occupa della realizzazione di un sistema di verifica mediante virtualizzazione tramite il quale è possibile fornire al dispositivo wearable dati di sensori precedentemente campionati al fine di testare gli algoritmi e massimizzarne l'efficienza. In particolare ci si concentrerà su problematiche relative ai tempi di trasmissione e si cercherà di adottare la strategia migliore al fine di simulare un sensore che al dispositivo wearable appaia esattamente come apparirebbero i suoi sensori *onboard*. Il lavoro effettuato durante la fase sperimentale corrente, come già accennato, si divide in due sezioni complementari. La prima, affrontata in questa tesi, è la realizzazione di un tool di virtualizzazione di cui si parlerà a breve. La seconda, affrontata invece nella tesi

complementare a questa, si occupa di gestire la routine a bordo del dispositivo wearable in modo tale che sia in grado di ospitare i dati ricevuti dal nodo fisso, interpretarli e scriverli in un'altra caratteristica dalla quale il nodo fisso li leggerà. Inoltre, il dispositivo wearable farà in modo che la virtualizzazione sia quanto più flessibile possibile, nel senso che permetterà dinamicamente la modifica dei parametri di configurazione senza modifiche dirette al codice. Ciò sarà possibile grazie alla definizione di messaggi con una struttura ben precisa che la routine a bordo della SensorTile saprà come intepretare.

Il tool che verrà realizzato servirà ad avviare una sessione di virtualizzazione settandone i parametri caratteristici e infine ad ottenere i dati sperimentali ottenuti, con i quali sarà possibile analizzare le prestazioni generali del sistema.

Poiché l'applicazione di virtualizzazione necessita di interfacciarsi col nodo fisso, e non è necessario che applicazione di virtualizzazione e nodo fisso si trovino all'interno della stessa rete locale, si è deciso di fare in modo che l'applicazione di virtualizzazione si possa collegare da remoto al nodo fisso. Per farlo sarà opportuno fornire l'indirizzo IP della rete all'interno della quale si trova il nodo e la porta sulla quale è in ascolto il server che gira sul nodo fisso. Anche in questo caso verranno in aiuto i socket, mediante la libreria *SocketIO*. È chiaro che non è esclusa la possibilità che l'applicazione di virtualizzazione possa essere eseguita all'interno della stessa rete su cui si trova il nodo fisso (Figura 20).

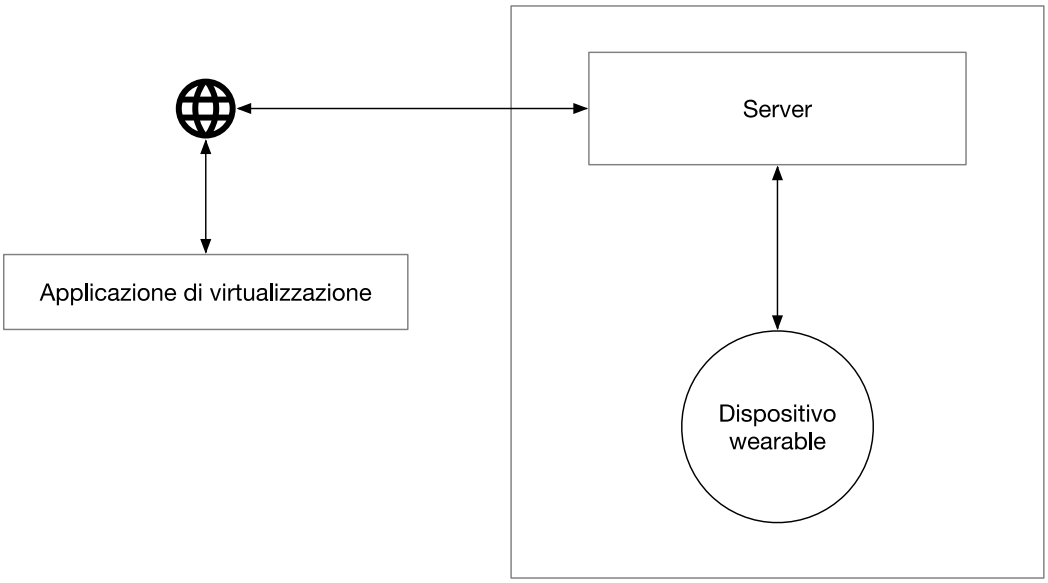

Figura 20 - Schema della comunicazione tra i dispositivi interoperanti nel tool di virtualizzazione

Per la realizzazione dell'interfaccia utente dell'applicazione di virtualizzazione è stato scelto l'utilizzo di ReactJS, una libreria di JavaScript ad alto livello che si basa sulla creazione di componenti dell'interfaccia grafica e al loro utilizzo mediante una sintassi che ricorda molto quella dell'HTML. Oltre a definire componenti personalizzati è possibile integrare librerie contenenti componenti realizzati da terze parti e creare interfacce tramite le usuali tecniche di *web styling* come le flexbox. La scelta di ReactJS per la realizzazione del tool di virtualizzazione è puramente arbitraria e non è orientata a motivi di efficienza poiché le elaborazioni avverrebbero comunque tra dispositivo fisso e dispositivo wearable. L'interfaccia utente servirà infatti solo per il caricamento del dataset attraverso il dispositivo remoto e per il settaggio dei parametri di configurazione del test di virtualizzazione. Vedremo a breve nel dettaglio lo scambio di tali

informazioni e descriveremo il funzionamento dell'intero sistema di virtualizzazione.

Per l'implementazione del meccanismo di virtualizzazione è stata posta particolare attenzione a problematiche legate alla comunicazione wireless. Una delle prime domande a cui si voleva dare una risposta era relativa a quanto potesse essere efficiente una virtualizzazione con un canale di comunicazione wireless. Per virtualizzare infatti ci sono due possibili approcci. Il primo è quello in cui il nodo fisso si occupa di gestire la temporizzazione, quindi di scandire l'intero dataset e inviarlo campione per campione alla frequenza di campionamento. In questo modo sorgono forti dubbi sulle performance di virtualizzazione a causa delle problematiche legate alla velocità di trasmissione nelle reti wireless. Per eliminare tale problematica si potrebbe pensare di instaurare un buffer nel dispositivo wearable, il quale stavolta gestirà la temporizzazione. In questo caso è inevitabile un offset di tempo iniziale durante il quale il dispositivo fisso comincia ad inviare il dataset in modo che il buffer del dispositivo wearable si popoli un po' prima di iniziare la virtualizzazione. Anche se la temporizzazione è gestita dal nodo wearable è necessario che il nodo fisso stabilisca anch'esso un *rate* di invio consono. Se il *rate* di invio da parte del nodo fisso è troppo basso, il dispositivo wearable probabilmente ad un certo punto troverà il buffer vuoto e non potrà portare a termine la virtualizzazione. Se il *rate* di invio, al contrario è troppo alto, si rischia di saturare il buffer del dispositivo wearable. Il giusto compromesso per evitare problemi come quelli appena citati è quello di inviare i dati alla stessa frequenza con cui il dispositivo wearable li preleva dal buffer.

Nel seguito ci soffermeremo con particolare attenzione sul primo tipo di approccio poiché, almeno dal lato del dispositivo fisso, è simile se non identico al secondo. L'unica differenza è il modo in cui il dispositivo wearable preleva i dati inviati.

Il meccanismo di virtualizzazione verrà sviluppato utilizzando due caratteristiche *BLE* che il dispositivo wearable espone. La prima, che chiameremo *VirtualSampleIn*, sarà quella in cui il nodo fisso andrà a scrivere il singolo campione del dataset. La seconda, che chiamiamo invece *VirtualSampleOut*, sarà la caratteristica alla quale il nodo fisso si sottoscriverà e verrà utilizzata per ricevere indietro i campioni inviati. È stato infatti implementato un comportamento che si è scelto di chiamare *Mirror Behavior* tramite il quale il dispositivo fisso può ricevere esattamente gli stessi dati inviati al fine di trarre conclusioni in merito al tempo che un singolo campione impiega dall'essere inviato all'essere ricevuto (*RTT – Round Trip Time*). Analizzeremo tale comportamento nel dettaglio più avanti.

4.2 Struttura dei messaggi scambiati

Prima di procedere con qualsiasi altra cosa è necessario innanzitutto capire che tipo di struttura devono avere i messaggi scambiati tra i due nodi. Per far ciò è fondamentale effettuare un'analisi delle informazioni che devono transitare da un nodo all'altro al fine di realizzare un sistema quanto più flessibile possibile, ovvero configurabile soltanto dal tool con cui un ipotetico utente avrebbe a che fare. In questo modo la virtualizzazione potrà essere configurata e portata a termine mediante

l'interfaccia grafica e non sarà necessario andare a modificare frammenti di codice né nel nodo fisso, né in quello wearable.

Un generico messaggio deve in primis contenere la tipologia. Nel caso dell'applicazione di virtualizzazione, sono stati individuati due tipi di messaggi differenti che la SensorTile deve poter ricevere: messaggi di configurazione e messaggi ordinari, ovvero quelli contenenti i dati da virtualizzare. Entrambi i tipi messaggio hanno una struttura suddivisa in byte.

Di seguito (Figura 21) si riporta la struttura di un messaggio di configurazione.

Figura 21 - Struttura di un messaggio di configurazione

Il messaggio di configurazione utilizza 6 byte dei 20 a disposizione. Il primo byte viene utilizzato per identificare la tipologia del messaggio e conterrà il valore 0 nel caso di messaggio di configurazione, ed 1 nel caso di messaggio ordinario (per ordinario s'intende il messaggio contenente il dato da virtualizzare). La dimensione del payload indica il numero di byte del messaggio ordinario che saranno dedicati ai dati da virtualizzare escluso il byte di configurazione. Il numero di campioni indica quanti campioni di un singolo test dovranno essere presi in considerazione. Il numero di ripetizioni consentirà invece di settare correttamente quante volte si vuole ripetere consecutivamente il singolo test del dataset.

Riportiamo invece adesso (Figura 22) la struttura di un messaggio ordinario per l'applicazione esaminata.

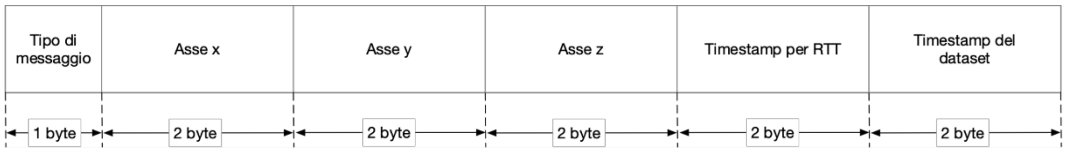

Figura 22 - Struttura di un messaggio ordinario

Nel caso specifico dell'applicazione distribuita per il rilevamento delle cadute, un messaggio ordinario è definito in modo tale che il suo payload sia esattamente di 10 byte. Tralasciando il byte di configurazione che è già stato descritto in precedenza, analizziamo il payload. I primi 6 byte del payload contengono i valori dei tre assi dell'accelerometro. Una cosa curiosa da notare è la presenza di due diversi tipi di timestamp. Quello che in figura appare come *Timestamp per RTT* rappresenta il numero di millisecondi passati dall'inizio di ogni test all'invio dell'*i-esimo* campione. Tale valore sarà quindi sempre crescente campione dopo campione. Questo timestamp, verrà utilizzato per calcolare l'*RTT*. Il nodo fisso, all'invio di ogni test, avvia un timer e quando un generico campione viene inviato mette il valore del timer all'interno del campo timestamp. Il dato viene scritto nella caratteristica *VirtualSapmleIn* del dispositivo wearable il quale dopo averlo letto lo scrive nella caratteristica *VirtualSampleOut* alla quale il nodo fisso è sottoscritto. Ciò scatena una notifica tramite la quale il nodo fisso leggerà il valore della caratteristica, compreso il timestamp che aveva precedentemente inserito come dato del campione. Il nodo fisso a questo punto potrà controllare il valore attuale del timer, e farne la differenza con quello presente all'interno del campione. In questo modo si riesce ad

ottenere quanto tempo è passato dall'invio del campione, alla sua completa ricezione. È proprio questo il fulcro del discorso in merito al *Mirror Behavior* accennato precedentemente. Il timestamp del dataset dovrebbe invece contenere un'informazione di tempo che in generale i dataset forniscono assieme ai dati raccolti. Poiché il dataset utilizzato per condurre l'esperimento non presenta alcuna informazione legata al tempo, si è scelto di riempire tale campo con il valore ideale in millisecondi di temporizzazione. Ciò significa che, poiché il sensore utilizzato per registrare il dataset presenta una frequenza di campionamento di 200Hz, il campo *Timestamp del dataset* conterrà il valore 0 per il primo campione, 5 per il secondo, 10 per il terzo e così via. L'informazione appena descritta non sarà determinate per l'applicazione in questione, ma sarà fondamentale nel caso in cui la temporizzazione deve avvenire a bordo della SensorTile.

4.3 Descrizione del funzionamento del sistema

La Figura 23 mostra il ciclo di funzionamento generale del sistema realizzato per la virtualizzazione. È presente una fase di scrittura della caratteristica *VirtualSampleIn* e una fase di lettura di *VirtualSampleOut* mediante notifca da parte del nodo fisso.

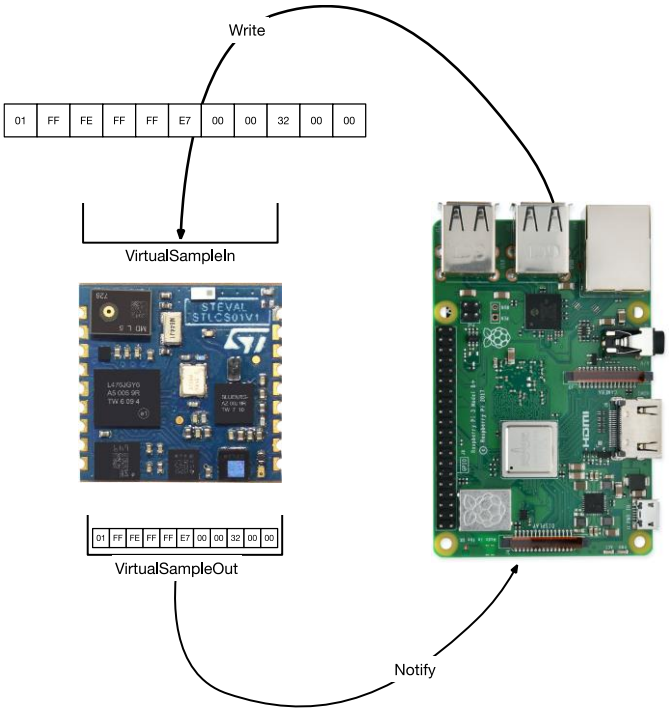

Figura 23 - Ciclo di lettura/scrittura tra Raspberry e SensorTile

Come detto precedentemente è stato realizzato un tool con interfaccia grafica per generare i messaggi appena visti, quindi configurare i parametri di virtualizzazione e dare inizio alla virtualizzazione stessa. La prima cosa da effettuare è la connessione al dispositivo wearable tramite il dispositivo fisso, del quale è necessario conoscere l'indirizzo IP e la porta sulla quale si trova in ascolto il programma server. Come è possibile notare dalla Figura 24, andranno compilati i due campi di testo che permetteranno la connessione della macchina remota al dispositivo fisso, e al dispositivo fisso di connettersi al dispositivo wearable tramite il suo indirizzo MAC.

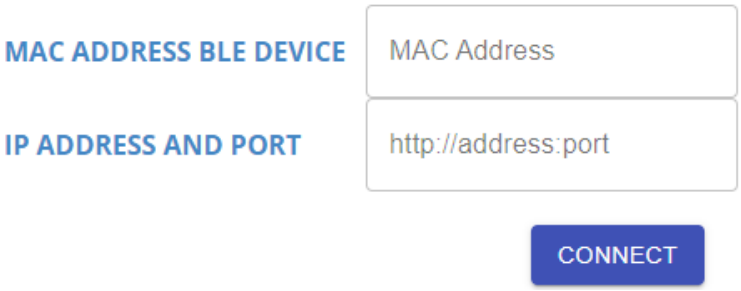

Figura 24 - Sezione del tool per la configurazione dei parametri di connessione

Una volta effettuata la connessione sarà data la possibilità di impostare i parametri di virtualizzazione (Figura 25).

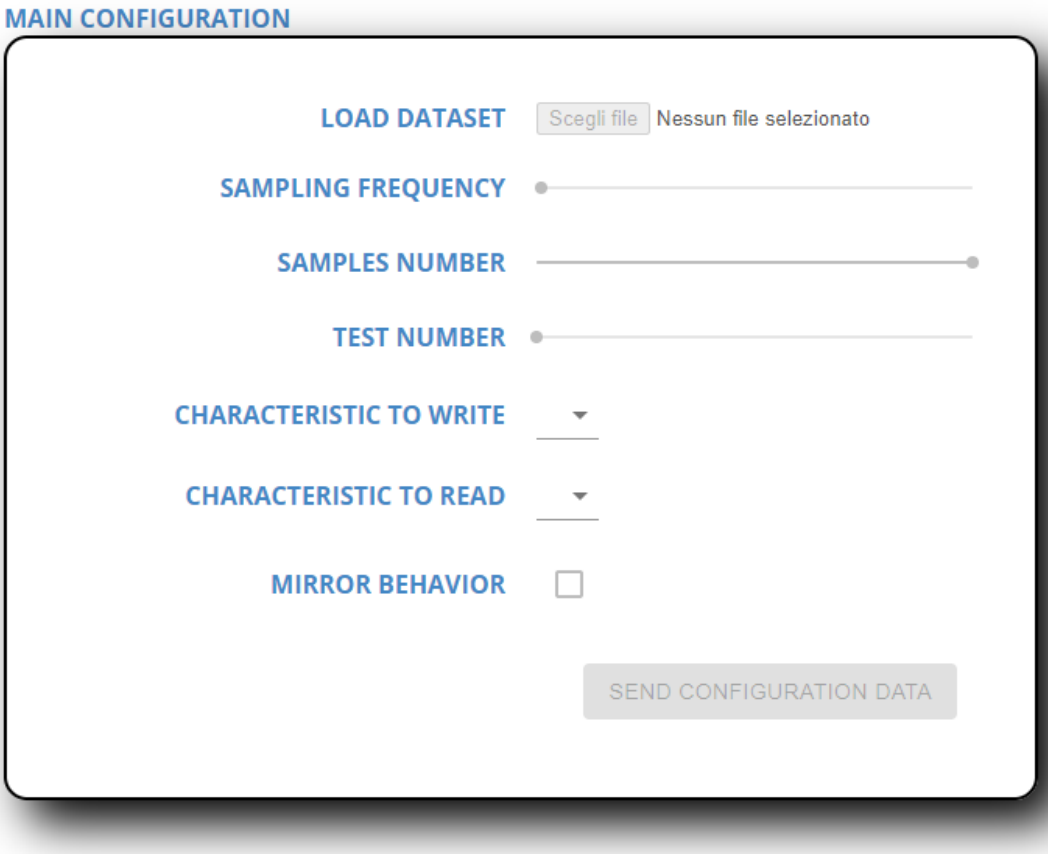

Figura 25 - Sezione del tool per la configurazione delle impostazioni di virtualizzazione

Tramite l'interfaccia in questione sarà possibile caricare un file del dataset contenete nel nostro caso un test di caduta. Sarà possibile impostare la frequenza di campionamento che deve simulare il sensore virtuale, il numero di campioni che devono essere considerati a partire dal primo campione del dataset caricato e il numero di ripetizioni consecutive che devono essere effettuate per un singolo test. Sarà inoltre possibile scegliere da un menù di selezione la caratteristica su cui scrivere il dataset (*VirtualSampleIn)* e la caratteristica da cui leggere nel caso in cui venga abilitato il *Mirror Behavior (VirtualSampleOut).*

Dopo aver effettuato le configurazioni, tramite il pulsante *Send Configuration Data* il dispositivo fisso comunicherà al dispositivo wearable i dati per la sessione di virtualizzazione. A questo punto sarà possibile premere un altro pulsante per avviare la virtualizzazione, al termine della quale verranno memorizzati dei file contenenti i risultati ottenuti.

In particolare, per la sessione di virtualizzazione, si è scelto di simulare per 100 volte consecutive uno dei test del dataset impostando la configurazione in modo che fosse equivalente a quella con la quale il dataset è stato registrato. È stata quindi impostata una frequenza di campionamento di 200Hz e un numero di campioni pari a 3000 (tutti i campioni di un singolo test). Per ricavare informazioni riguardo l'*RTT* è stato abilitato il *Mirror Behavior*.

4.4 Risultati sperimentali

I risultati sperimentali ottenuti possono considerarsi abbastanza soddisfacenti. Considerando il fatto che i campioni da inviare sono in tutto 3000, che un campione deve essere inviato ogni 5ms e che il primo campione è inviato all'istante di tempo 0, l'invio di tutti i campioni per un singolo test dovrebbe completarsi esattamente in 14.995s. Per ognuno dei 100 test consecutivi, è stato calcolato il tempo impiegato. Dei tempi ottenuti ne è stata effettuata una media e calcolata la varianza. È stato inoltre calcolato l'*RTT* per ogni campione inviato e per ognuno dei 100 test è stato calcolato un *RTT* medio. Gli *RTT* medi sono stati a loro volta mediati ulteriormente al fine di ottenere un'indicazione di *Round Trip Time* complessiva. Di seguito viene riportata una tabella riassuntiva dei dati numerici ottenuti (Figura 26).

μ [s] *(Valor medio)*	15.11
σ^2 *(Varianza)*	0.00034
RTT *[ms]*	24.64

Figura 26 - Tabella riassuntiva risultati sperimentali

Come è possibile notare il valor medio si discosta soltanto di 0.115s dal valore atteso. Ciò significa che è stato impiegato un tempo maggiore rispetto a quello ideale, ma soltanto dello 0.77% in più. Il valore molto basso

della varianza conferma inoltre che il tempo impiegato da ciascuno dei 100 test, non si discosta molto dal valor medio. Il grafico seguente (Figura 27) mostra a tal proposito l'andamento del valor medio all'aumentare del numero di test.

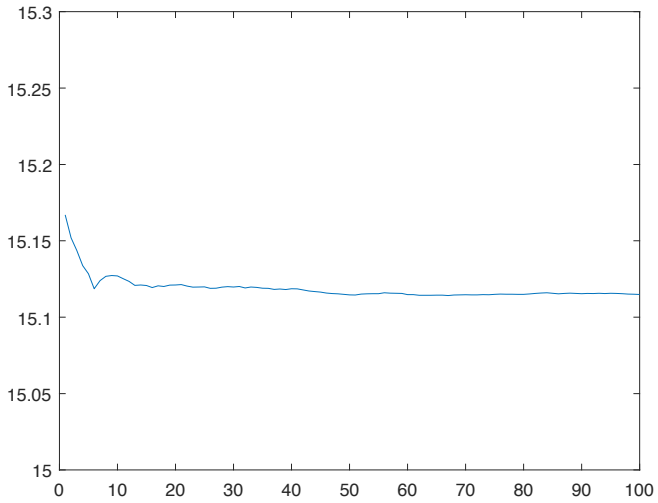

Figura 27 - Curva rappresentante il valor medio in funzione del numero di test effettuati

Se inizialmente il valor medio sembra oscillare, subito dopo una decina di test questo comincia ad assestarsi sempre più attorno ad un valore ben definito che confluirà proprio al valor medio nella centesima ascissa.

L'*RTT* fornisce la reattività del sistema durante la fase in cui il nodo wearable scrive la caratteristica *VirtualSampleOut* e il nodo fisso la legge. Nel caso di questa particolare prova sperimentale i dati del *RTT* forniscono un'importante informazione in merito ai tempi di risposta del sistema. Se piuttosto che ricevere continuamente dati indietro, il dispositivo fisso si mettesse in ascolto di un messaggio di allarme sempre attraverso *VirtualSampleOut*, il tempo di ricezione dell'allarme a partire dal rilevamento dell'emergenza da parte del dispositivo wearable sarebbe al massimo soltanto nell'ordine della ventina di millisecondi. Un tempo di

quest'ordine di grandezza può considerarsi pienamente soddisfacente per applicazioni di rilevamento di eventi in real time.

5 Conclusioni

I risultati ottenuti dai test sperimentali mostrano come la virtualizzazione riesce a fornire un supporto efficiente alla realizzazione di applicazioni distribuite complesse per smart environment. Nonostante i limiti fisici imposti dalle reti wireless si è visto come una sessione virtualizzata possa essere comparata ad una sessione di rilevamento con sensori reali. L'overhead di tempo dovuto all'elaborazione può considerarsi piccolo a tal punto da rendere efficiente la virtualizzazione. Sebbene non sia stato affrontato a fondo l'approccio della temporizzazione a bordo del dispositivo wearable, i dati ottenuti mediante temporizzazione da parte del nodo fisso fanno ben sperare per approcci di virtualizzazione in cui la temporizzazione è affidata al nodo sensore. Nonostante le ottime prestazioni in merito alle tempistiche ottenute dai risultati sperimentali mediante virtualizzazione temporizzata dal nodo fisso, nei casi in cui si vuole virtualizzare una WSN complessa, tali prestazioni potrebbero non essere soddisfacenti allo stesso modo. Temporizzare a bordo del nodo fisso non è infatti una scelta ottimale dal punto di vista della scalabilità poiché gestire la temporizzazione all'aumentare dei nodi potrebbe comportare notevoli ritardi che falsificherebbero i risultati della virtualizzazione stessa. In questo caso, sicuramente, un approccio più corretto sarebbe quello di allocare in ogni nodo sensore un buffer il quale potrebbe presentare una struttura di tipo FIFO. In questo modo il nodo fisso comincerà a precaricare dei buffer per fare in modo che i dati da virtualizzare siano direttamente disponibili per i nodi sensore, i quali dovranno occuparsi soltanto di prelevarli ad un *rate* stabilito ed elaborarli.

Una caratteristica soddisfacente del sistema realizzato è sicuramente quella della realizzazione di applicazioni per il rilevamento di eventi in tempo reale. Il valore del *Round Trip Time* ha mostrato come il tempo di invio e ricezione di uno stesso dato risulti particolarmente basso all'interno della rete nonostante si stia lavorando con dispositivi le cui risorse computazionali non sono eccessive. È poi vero che informazioni come ad esempio le notifiche devono transitare tramite internet dal nodo fisso al *caregiver*, ma ciò non preoccupa, quantomeno dal punto di vista dei tempi di trasmissione, date le prestazioni delle reti attuali.

Uno dei vantaggi principali della virtualizzazione è dato sicuramente dalla possibilità di testare le WSN con dati preesistenti e integrare pian piano sensori reali al posto di quelli virtuali. In questo modo si avrà a che fare con delle reti ibride [17] che vedranno la cooperazione tra sensori reali e virtuali fornendo un livello di astrazione tale da rendere indistinguibili gli uni dagli altri agli occhi dell'applicazione. I lavori futuri su cui ci si concentrerà vedranno in primo luogo l'estensione dell'approccio esaminato a reti di sensori più estese, cercando come obiettivo di ottenere tempistiche simili a quelle ottenute dagli esperimenti effettuati durante lo sviluppo di questo progetto di tesi. Un'altra caratteristica da aggiungere al sistema complessivo sarà sicuramente quella di un meccanismo basato sullo scambio di codice eseguibile tra dispositivi di una rete di sensori dalle risorse limitate (DC4CD). A seguito della futura realizzazione di meccanismi di verifica per la costruzione di WSN virtualizzate o ibride più complesse, sarebbe interessante testare il sistema in situazioni reali. In particolare, i futuri sviluppi vedranno l'implementazione sullo smartphone del caregiver di un'applicazione mobile capace di notificare situazioni di emergenza rilevate dal sistema remoto. Verranno inoltre affrontate più a

fondo tematiche riguardanti la privacy degli assistiti al fine di mantenere in sicurezza i dati raccolti e far sì che l'invio di dati personali tramite internet avvenga in modo sicuro e misurato, nel senso che si cercherà di capire quali dati è strettamente necessario far transitare in internet e quali invece no.

Bibliografia

[1] A. De Paola, P. Ferraro, S. Gaglio, G. Lo Re, M. Morana, M. Ortolani and D. Peri, "An Ambient Intelligence System for Assisted Living," in *2017 AEIT International Annual Conference (2017 AEIT)*, Cagliari, 2017.

[2] A. De Paola, P. Ferraro, S. Gaglio, G. Lo Re, M. Morana, M. Ortolani and D. Peri, "A Context-aware System for Ambient Assisted Living," in *11th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2017)*, Philadelphia, 2017.

[3] S. Gaglio, G. Lo Re, G. Martorella and D. Peri, "High-level programming and symbolic reasoning on IoT resource constrained devices," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST,* vol. 150, pp. 58-63, 2015.

[4] S. Gaglio, G. L. Re, G. Martorella and D. Peri, "DC4CD: A Platform for Distributed Computing on Constrained Devices," *ACM Trans. Embed. Comput. Syst.,* vol. 17, p. 27:1–27:25, 12 2017.

[5] S. Gaglio, G. L. Re, G. Martorella and D. Peri, "A Lightweight Middleware Platform for Distributed Computing on Wireless Sensor Networks," *Procedia Computer Science,* vol. 32, pp. 908-913, 2014.

[6] S. Gaglio, G. L. Re, G. Martorella and D. Peri, "WSN Design and Verification using On-board Executable Specifications," *IEEE Transactions on Industrial Informatics,* pp. 1-1, 2018.

[7] S. Gaglio, G. L. Re, G. Martorella and D. Peri, “A fast and interactive approach to application development on Wireless Sensor and Actuator Networks,” 2014.

[8] D. Peri, “Body Area Networks and Healthcare,” in *Advances onto the Internet of Things: How Ontologies Make the Internet of Things Meaningful*, S. Gaglio and G. Lo Re, Eds., Cham, Springer International Publishing, 2014, p. 301–310.

[9] H. Sundani, V. Li, M. Devabhaktuni, P. Alam, Bhattacharya, H. Sundani, H. Li, V. Devabhaktuni, M. Alam and P. Bhattacharya, “Wireless Sensor Network Simulators: A Survey and Comparisons,” *International Journal Of Computer Networks,* vol. 2, 1 2011.

[10] “Bluetooth low energy Characteristics, a beginner's tutorial,” 18 Mar 2016. [Online]. Available: https://devzone.nordicsemi.com/nordic/short-range-guides/b/bluetooth-low-energy/posts/ble-characteristics-a-beginners-tutorial.

[11] L. Leonardi, G. Patti and L. Lo Bello, “Multi-hop Real-time Communications over Bluetooth Low Energy Industrial Wireless Mesh Networks,” *IEEE Access,* vol. PP, pp. 1-1, 5 2018.

[12] “STEVAL-STLKT01V1 - Sensortile development kit,” [Online]. Available: https://www.st.com/resource/en/data_brief/steval-stlkt01v1.pdf.

[13] “Raspberry Pi 3 Model B+ - The final revision of our third-generation single-board computer,” [Online]. Available:

https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus.

[14] "SocketIO Official Documentation," [Online]. Available: https://socket.io/docs/.

[15] D. Coleman, "A Nodejs BLE (Bluetooth Low Energy) central module," [Online]. Available: https://github.com/noble/noble.

[16] "SISTEMIC: SisFall Dataset," [Online]. Available: http://sistemic.udea.edu.co/en/investigacion/proyectos/english-falls.

[17] S. Saginbekov and C. Shakenov, *Testing Wireless Sensor Networks with Hybrid Simulators,* 2016.