ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI UNA PIATTAFORMA SOFTWARE PER APPLICAZIONI DI MONITORAGGIO DI PARAMETRI VITALI BASATE SU TECNOLOGIE WEARABLE E MOBILE

Relazione finale in
PROGRAMMAZIONE DI SISTEMI EMBEDDED

Relatore
Prof. ALESSANDRO RICCI

Co-relatore
Ing. ANGELO CROATTI

Presentata da
DAVIDE FOSCHI

Seconda Sessione di Laurea
Anno Accademico 2014 – 2015

PAROLE CHIAVE

communication

wearable computing

health care

single-board computer

sensoristic

"I started connecting things to my body during my childhood. I approached the computer as a mediating element, as a form of visual art." - Steve Mann

Indice

Introduzione

Negli ultimi 30 anni la tecnologia wearable computing ha subito un incremento tecnologico esponenziale, arrivando a fondersi con la vita di tutti i giorni. Oggigiorno dispositivi wearable come smartphone e smartwatch ricoprono un ruolo fondamentale nelle mansioni della maggior parte della popolazione. La versatilità di questa tecnologia permettere di applicarla a diversi contesti, migliorandone le caratteristiche e offrendo una esperienza migliore. Soprattutto negli ultimi anni, aziende e ricercatori hanno cercato di semplificare la realizzazione di sistemi wearable, sviluppando tecnologie economiche che permettessero a chiunque di costruirsi il proprio prototipo di dispositivo wearable senza spendere ingenti somme di denaro. Fra le tante applicazioni del wearable computing, una delle più importanti è senz'altro l'healthcare, da cui sono stati commercializzati importanti dispositivi per la cura del proprio stato di salute.

L'obiettivo di questa tesi è proporre una piattaforma a supporto dello sviluppo di applicazioni mobile per il monitoraggio dei parametri vitale di un utente, attraverso l'utilizzo di tecnologie wearable. La tesi è strutturata in 7 capitoli.

Il primo capitolo presenta lo stato dell'arte inerente al wearable computing, introducendo alcuni esempi di applicazioni in ambito healthcare e non. Inoltre dedica un paragrafo alla prototipazione del weararable.

Nel secondo capitolo viene descritto il caso di studio, analizzando i tutti i requisiti funzionali e non funzionali da rispettare.

Nel terzo e quarto capitolo vengono modellati il dominio applicativo, discusse le scelte tecnologiche e descritta tutta la fase di progettazione del sistema, sottolineando problematiche incontrate e soluzioni adottate.

Gli ultimi 3 capitoli mostrano scelte implementative adottate, concentrandosi sulle componenti più critiche del sistema, una fase di collaudo del sistema per verificare il comportamento della piattaforma e infine vengono espresse le valutazioni finali sul progetto finito.

Segue infine una breve conclusione e un accesso a sviluppi futuri.

Capitolo 1

Wearable Computing

Il wearable computing indica un gruppo di dispositivi computazionali, miniaturizzati per essere indossati, che assistono l'utilizzatore in determinate circostanze. Essi possono essere posizionati sopra o sotto gli abiti, oppure essere a loro volta dei vestiti. Chi utilizza un wearable device ha a disposizione uno strumento che permette di comunicare con altre persone e aumenta la visione della realtà e permette di ottenere maggiori informazioni da ciò che lo circonda. Si ottiene una esperienza di vita aumentata da tutto ciò che il dispositivo riesce a rilevare. Si crea quella che viene chiamata la realtà aumentata (augmented reality, AR) che mischia elementi reali con elementi virtuali.

Lo scopo del wearable computing è quello di unire le capacità intellettive di una persona con le capacità di calcolo di un computer per raggiungere quello che è definito come Humanistic Intelligence, ovvero una intelligenza superiore ottenuta unendo il pensiero umano al ciclo computazionele di un computer.

1.1 Cenni storici

Wearable computing può essere interpretato in diversi modi, per cui è difficile indentificare la nascita di esso: se si interpreta computing come semplice capacità di eseguire dei calcoli, allora i primi dispositivi di questo settore sono gli abachi ad anello e gli orologi da taschino inventati nel sedicesimo secolo. Nel 1961 i matematici Edward O. Thorp e Claude Shannon costruirono un dispositivo per barare alla roulette: constisteva in un computer analogico nascosto in una scarpa che prevedeva l'ottante in cui sarebbe fermata la sfera, inviando un segnale acustico all'utilizzatore. Questo è considerato il predeccessore di tutti i wearable computers.

Figura 1.1: Dispositivo wearable utilizzato per barare alla roulette

Negli anni '70 e '80 vengono costruite le basi del wearable computing da Steve Mann. Tra le sue invenzioni sono da citare la Wearable Wireless Webcam e il WearComp, un sistema wearable composto da un computer general-purpose adattato alle dimensioni di uno zaino e da un head-mounted display visibile su un solo occhio. La tastiera era sostituita da un keyer, un manipolatore a una mano derivato dal telegrafo. Mann portò le sue invenzioni al MIT nel 1991, dove contribuì alla creazione del MIT Wearable Computing Laboratory insieme a un altro pioniere del mondo wearable: Thad Starner. Anche Thad sviluppò nel 1993 un sistema wearable simile a quello di Mann: basò il suo sistema sul wearable computer chiamato Hip-PC progettato da Doug Platt e Gerald Maguire nel 1990 e sul display prodotto dalla Reflection Technology chiamato Private Eye. Quest'ultimo dispositivo sarà poi utilizzato da Steve Feiner per creare un sistema che offre realtà aumentata: KARMA. Nel frattempo nasce nel 1990 la Xybernaut Corporation, la prima azienda che inizia a vendere wearable devices al pubblico.

Negli anni novanta vennero istituite le prime conferenze centrate sul wearable computing, come la International Symposium on Wearable Coumputing proposta da Mann. Nella seconda metà degli anni '90 avviene un cambiamento nel design dei dispositivi. L'obiettivo ora è quello di racchiudere display e computer in un unico device: nascono cosi i primi computer da polso.

Negli anni 2000 il grande progresso tecnologico porta allo sviluppo di dispositivi sempre più performanti. Il mondo del wearable computing inizia a diffondersi massivamente nella popolazione e diventa uno strumento per migliorare la qualità della vita di tutte le persone. Nascono così i primi lettori mp3, smartphones, oggetti di comune utilizzo. Ai giorni nostri si trovano le evoluzioni degli occhiali utilizzati negli '80, che sono diventati dei dispositivi

indipendenti, che incorporano una unità di calcolo, e che vengono comandati da comandi vocali. Un esempio sono i google glass rilasciati sul mercato americano nel 2014. Il wearable computing è diventato uno strumento di comune utilizzo che migliora la qualità della vita.

1.2 Caratteristiche

Un wearable computer può essere descritto come un computer funzionale, autoalimentato e autosufficiente che viene indossato sul proprio corpo, fornendo l'accesso ad interazioni con informazioni provenienti dall'ambiente circostante in ogni momento.

Fisicamente il wearable computer consiste in un dispositivo di ridotte dimensioni a cui sono collegati diversi sensori a seconda delle funzionalità per cui è costruito. Le informazioni ottenute sono visualizzabili dall'utilizzatore del wearable computer attraverso una interfaccia video, che principalmente è costituita da un piccolo schermo posto davanti ad un occhio o da uno schermo portatile.

Per ottenere le caratteristiche descritte in precedenza, il computer deve essere come un aiutante. Egli condivide le esperienze dell'utente e impara come egli ragiona e comunica con l'ambiente. In questo modo può fornire una assistenza sempre migliore ed adattarsi alle necessità dell'utilizzatore. Generalizzando, un wearable computer dovrebbe avere i seguenti attributi.

Persistenza e accesso costante alle informazioni. Un wearable deve essere costruito per lavorare continuamente nel corso della giornata e deve poter interagire con l'utente in qualunque momento, interrompendosi quando necessario e appropriato. Viceversa, l'utente può accedere al dispositivo rapidamente e con poco impegno. Inoltre tale dispositivo deve essere indossabile senza che risulti un impedimento per l'utente.

Riconoscere e modellare l'ambiente. Per conferire all'utente il migliore supporto cognitivo, il wearable deve saper osservare e modellare l'ambiente che lo circonda e lo stato fisico e mentale dell'utente. In alcuni casi, l'utente stesso potrebbe specificare alcuni dettagli per aiutare il dispositivo nei suoi compiti. Il wearable dovrebbe informare l'utente del suo stato attuale mediante display o tecnologie simili e rendere le informazioni raccolte dall'ambiente osservabili dall'utente.

Adattare le modalità di interazione in base alla situazione dell'utente. Il dispositivo dovrebbe adattare il proprio input e output automaticamente in base alla situazione in cui l'utente si trovo in un determinato momento. Per esempio, nel momento un cui l'utente partecipa ad una conferenza, il dispositivo potrebbe comunicare le informazioni attraverso un display

montato davanti ad un occhio. Quando invece l'utente entra nella proprio automobile, il wearable potrebbe cambiare l'interfaccia video con una interfaccia audio. In ogni caso, l'interfaccia deve essere secondaria rispetto alle attività primarie svolte dall'utente e dovrebbe richiedere la minima attenzione possibile.

Aumentare e mediare le interazioni con l'ambiente. Il wearable dovrebbe fornire informazioni di supporto sia in campo virtuale che in quello reale. Per esempio, il dispositivo dovrebbe raccogliere in automatico informazioni e risorse rilevanti su un particolare oggetto fisico (ma che potrebbe essere anche un luogo), e filtrarle in base alle necessità attuali e alle preferenze dell'utente. In generale, il wearable dovrebbe fungere da mediatore fra ambiente e utente, presentando una interfaccia coerente e che rispecchi le preferenze dell'utente. Infine il dispositivo dovrebbe gestire eventuali interruzioni, quali chiamate telefoniche o e-mail, per servire al meglio l'utente.

Nello sviluppo di un wearable computer è importante rispettare gli attributi sopracitati, ma è anche importante tenere in considerazioni le problematiche più comuni a cui si può andare incontro nello sviluppo di questo genere di dispositivi.

- Costo: i wearable computers sono fra i pezzi di tecnologia più sofisticati ai giorni nostri. Sofisticato implica un costo elevato per via della tecnologia di alto livello necessario allo sviluppo. Il prodotto finale può risultare costoso e fuori dalla portata di molte persone,

- Peso: i wearable computers posso essere abbastanza pesanti a causa dei numerosi componenti che li compongono. Un wearable computer richiede una unità di controllo (CPU), monitor e un dispositivo per l'input. Quando gesti componenti sono separati fra di loro, possono essere scomodi da indossare e da portare per tempo prolungato. Il computer in questione può invece essere abbastanza pesante nel caso in cui tutti i componenti siano uniti nello stesso dispositivo,

- Scomodità: Durante le giornate calde o periodi di alta attività del wearable computer, esso tende a surriscaldarsi e a irritare l'utilizzatore. Questo avviene perché il computer emettere calore indipendentemente dal sistema di raffreddamento e l'utente emette calore a sua volta durante le sue mansioni giornaliere. Inoltre, un dispositivo wearable complesso può risultare scomodo da indossare per tempo prolungato,

- Sicurezza: se lasciati incustoditi, i wearable computers possono costituire una breccia nella sicurezza. Essi possono essere violati da coloro con cui l'utente entra in contatto. Considerando le aziende, i wearable computers

possono essere collegati al server principale per facilitare la comunicazione tra utente e compagnia stessa. Lasciare incustoditi tali dispositivi permettere a chiunque di rubare informazioni sensibili dall'azienda.

1.3 Applicazioni del wearable computing

Il wearable computing offre enormi possibilità per la creazione di nuovi sistemi. L'applicazione di questa tecnologia spazia in moltissime aree di interesse. Un esempio è la telefonia, in cui la tecnologia wearable ha permesso di aumentarne esponenzialmente le capacità di calcolo, trasformando un normale telefono in un mini-computer. Questi dispositivi, chiamati Smartphone, fanno parte ormai della vita della maggior parte delle persone.

Altri esempi di dispositivi wearable attualmente in commercio sono gli smartwatch. Oltre a svolgere il normale compito di un orologio, gli smartwatch permettono di leggere e-mail, collegarsi via bluetooth al proprio smartphone per ricevere notifiche o connettersi a internet. Inoltre possono incorporare dei sensori per misurare alcuni parametri vitali, quali la temperatura corporea o il battito cardiaco.

Figura 1.2: Sony Smartwatch

Un applicazione molto in voga negli ultimi anni riguarda la realtà aumentata. Quelli che negli '80 e '90 erano semplici display posti davanti ad un occhio, ora sono diventati uno strumento in grado di alterare la visione della realtà, andando a creare uno strato virtuale sovrapposto all'ambiente reale. Un esempio molto conosciuto è il GoogleGlass. Il dispositivo osserva gli elementi che

compongono l'ambiente circostante che li aumenta aggiungendo informazioni aggiuntive. Queste sono visibili attraverso il glass che sovrappone all'elemento i dati raccolti come se fosse una notifica.

Figura 1.3: GoogleGlass

Un altro esempio di realtà aumentata è l'applicazione Wikitude sviluppata per IPhone, che permette di ottenere informazioni aggiuntive di un elemento osservandolo attraverso la fotocamera del tuo IPhone.

Infine, una applicazione molto importante del wearable computing è senz'altro l'healthcare che sarà approfondita nel prossimo paragrafo.

1.4 Sistemi wearable per l'healthcare

Fra i numerosi utilizzi del wearable computing, un campo molto esteso in cui viene applicata questa tecnologia è l'healthcare. Innanzitutto per healthcare si intende la diagnosi,trattamento e prevenzione di malattie, ferire e altri impedimenti fisici e mentali del corpo umano. La tecnologia wearable per applicazioni healthcare ha attratto l'attenzione di ricercatori e business communities (aziende e uomini d'affari) fin dai primi anni '90. Questo interesse non è solo dovuto al rapido avanzamento della tecnologia, come ad esempio nuovi sensori, miniaturizzazione dell'elettronica e maggiore capacità di calcolo, ma anche dal bisogno di migliorare la qualità e l'efficienza dell'assistenza sanitaria in ambiente casalingo e non. I dispositivi wearable offrirebbero una alternativa economia per l'healthcare rispetto alle normali istituzioni sanitarie. La cura della propria salute deve essere accessibile a chiunque, a basso costo e in qualunque momento di necessità.

Le caratteristiche di un sistema wearable permettono di superare le limitazioni dei dispositivi medici convenzionali e di fornire un feedback in real-time di uno o più parametri vitali rivelati per un arco di tempo prolungato. Alcuni esempi pratici sono dispositivi per l'elettrocardiogramma (ECG), pressione

sanguigna, respirazione e temperatura. Questi dati sono rilevati sul paziente e sull'ambiente circostante con l'ausilio di sensori integrati negli abiti o indossati direttamente (textile). I valori rilevati sono trasmessi wireless o wired a un dispositivo portatile o a un database che archivia e processa i dati. I risultati possono poi essere inoltrati a strutture specializzate per ulteriori analisi. In generale questi sistemi wearable permettono ai pazienti sono controllare i propri parametri vitali senza la necessità di recarsi in strutture specializzate e alleggeriscono la mole di lavoro dei professionisti.

Al giorno d'oggi sono stati raggiunti obiettivi importanti in questo settore e sono stati sviluppati dispositivi wearable innovativi che racchiudono l'essenza dell'healthcare. Principalmente questi dispositivi si dividono in due categorie: sistemi non tessili e smartclothing. Di seguito verranno presentati alcuni esempi di sistemi sviluppati negli ultimi due decenni.

Considerando una condizione di salute critica, AMON è un dispositivo wrist-worn che monitora pazienti con problemi respiratori e cardiaci. Esso rileva ed elabora i parametri vitali, individua situazioni di emergenza autonomamente e trasmette in maniera wireless tutte le informazioni a un centro medico.

Figura 1.4: Prototipo del dispositivo AMON

Un dispositivo interessante, che inoltre sfrutta le capacità di uno smartphone, è HealthGear. Questo sistema monitora la pressione e la saturazione del sangue e rileva automaticamente fenomeni di apnea durante il sonno. Fisicamente consiste in un sensore non invasivo collegato via bluetooth con lo smartphone che raccoglie,trasmette e analizza i dati rilevati e visualizza i risultati ottenuti su una interfaccia user friendly.

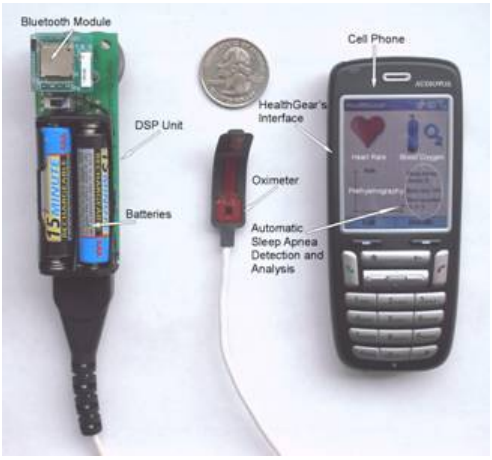


Figura 1.5: Sistema HealthGear

L'ultimo dispositivo preso in considerazione fa parte della categoria smart-clothing. MyHeart ha l'obiettivo di prevenire e monitorare malattie cardio-vascolari. L'hardware comprende l'utilizzo di elettronica avanzata applicata ad una maglietta, permettendo all'utente di controllare il proprio stato di salute. Il sistema non solo svolge funzioni di monitoraggio ma propone soluzioni adottabili dal paziente. Anche questo dispositivo consente sia un accesso ai dati locale e real-time oppure un accesso remoto per ottenere l'aiuto di professionisti.

Figura 1.6: Smartcloth MyHeart

1.5 Prototipazione del wearable

Lo sviluppo di sistemi wearable era sempre stato un compito molto arduo per via degli attributi da rispettare e delle problematiche da risolvere. Era necessaria una vasta conoscenza sono solo in campo informatico (software) ma anche in campo elettronico (hardware) e non solo. Nasceva la necessità, ma soprattutto il desiderio, di poter costruire sistemi wearable o più in generale embedded senza considerare tutti i fattori sopracitati e concentrandosi maggiormente sulle funzionalità offerte.

Verso la metà degli anni 2000 vengono introdotte sul mercato le prime tecnologie che permettono di prototipare piccoli sistemi wearable ed embedded in maniera agile ed efficiente. Venivano commercializzati i primi single-board computer (già costruiti negli anni '80) che però disponevano di numerosi - pin facilmente accessibili per il controllo di input e output sia analogico che digitale. Queste nuove schede general purpose sono programmabili utilizzando comuni linguaggi di programmazione (o derivati da essi) quali C++ e java e sono costruite rispettando gli attributi e le problematiche già elencate.

Ovviamente ogni scheda ha i propri vantaggi e svantaggi ed in seguito verranno presentate due delle famiglie di schede più utilizzate.

La prima presa in esame è la famiglia Arduino. Introdotta nel 2005, questo progetto ideato da un team italiano propone delle piattaforme per la prototipazione di sistemi embedded a basso costo e completamente open-source, sia livello hardware che livello software. Nello specifico verranno presentate le caratteristiche della scheda più venduta, ovvero Arduino Uno. L'unità computazionale è costituita dal microcontrollore ATmega328p con architettura Harvard a 8 bit e memoria con indirizzi a 16 bit incorporata. La scheda possiede 14 pin di input/output di cui 6 utilizzabili in PWM, 6 input analogici, un connettore USB e un jack per l'alimentazione (5V). Alcuni di questi pin permettono di utilizzare protocolli di comunicazione seriale quali I2C e SPI, mentre altri permettono la gestione dell'interrupt. Le limitazioni di questa scheda riguardano la capacità di calcolo, che non permette di costruire sistemi troppo complessi, e la memoria che non permette di utilizzare molte stringhe e chiamate ricorsive. Per quanto riguarda il software, tutte le schede Arduino utilizzano il framework Wiring che condivide molte delle caratteristiche del linguaggio C/C++. Arduino rende disponibile Arduino IDE come interfaccia per la programmazione e il caricamento di programmi sulla scheda, che avviene tramite un collegamento USB. è possibile caricare in memoria in singolo programma alla volta e il microcontrollore lo esegue ciclicamente finché rimane alimentato.

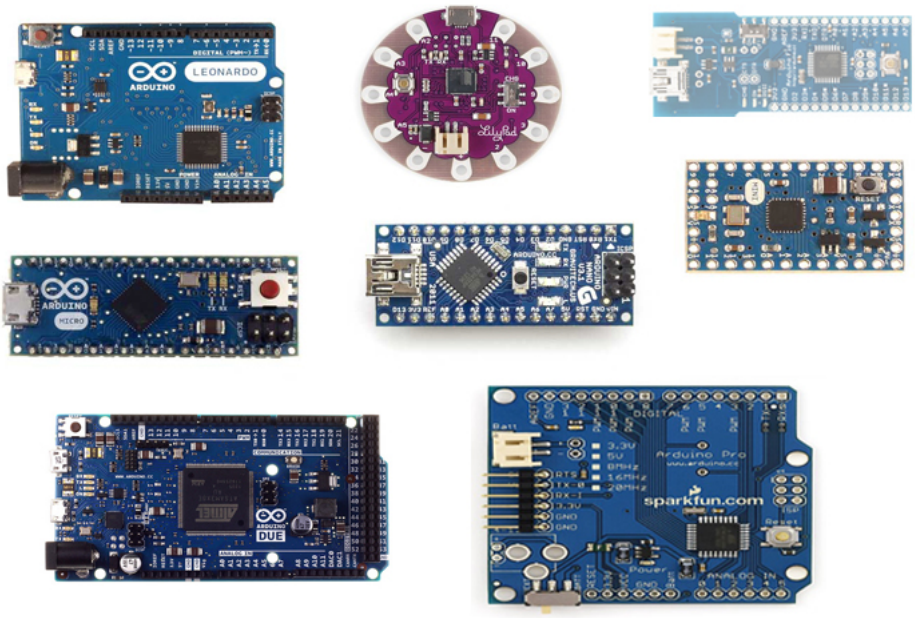


Figura 1.7: Famiglia Arduino

La seconda famiglia considerata è quella sviluppata da Raspberry PI Foundation. L'obiettivo era di quello sviluppare una scheda versatile per scopi didattici e oggi è una delle schede più utilizzare per la prototipazione di sistemi embedded. Per quanto riguarda l'hardware, esattamente come Arduino, vi sono dei pin dedicati all'input/output con possibilità di utilizzare protocolli seriali. La differenza principale riguarda l'unità di calcolo: Raspberry utilizza il chip BCM2835 che incorpora tutte le componenti tipiche di un normale computer, ovvero CPU, RAM e scheda video. Oltre agli ingressi presenti su Arduino, Raspberry presenta le classiche porte presenti sui computer, come HDMI, Ethernet, RCA e SD (o microSD a seconda della versione). Questo architettura permettere l'utilizzo di un sistema operativo. In questo caso viene utilizzato Raspbian, basato sulla distribuzione Debian. Per quanto riguarda il software, grazie al supporto del sistema operativo è possibile utilizzare linguaggi ad alto livello con Java: è possibile programmare su un normale computer utilizzando IDE molto conosciute come Ecplise ed eseguire il programma sul Raspberry utilizzando un collegamento USB o ethernet. Grazie alla presenza di un sistema operativo è possibile usufruire di tecniche di programmazione più avanzate.

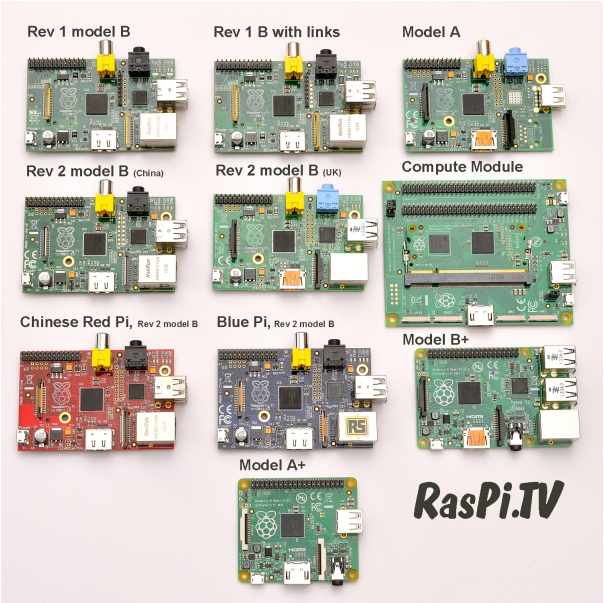


Figura 1.8: Famiglia RaspberryPI

Le applicazioni che queste schede offrono sono sconfinate ed un esempio riguarda healthcare introdotta precedentemente. Grazie alle loro caratteristiche hardware, queste schede si prestano molto bene come base per un sistema wearable per l'healthcare. Un esempio di questa applicazione è proposto dalla Cooking Hacks, che ha costruito un sistema indossabile per il monitoraggio dei parametri vitali basandosi su schede per la prototipazione molto comuni. La stessa azienda ha sviluppato una scheda aggiuntiva chiamata eHealth per la gestione dei sensori.

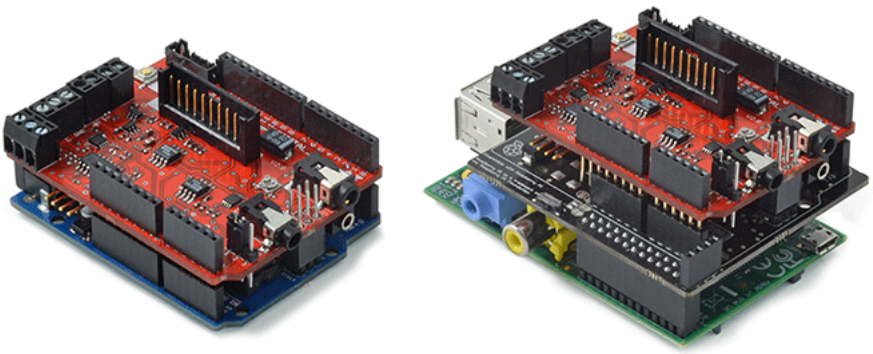


Figura 1.9: Utilizzo della scheda eHealth con Arduino e Raspberry

Capitolo 2

Caso di Studio e Analisi dei Requisiti

Come accennato nell'introduzione, l'idea di base è quella di realizzare una piattaforma a supporto dello sviluppo di applicazioni mobile in ambito healthcare.

Il sistema dovrà gestire un assortimento di sensori più o meno complessi, offrendo allo sviluppatore delle API per la lettura dei parametri vitali e l'opportunità di richiedere specifici valori di interesse.

Il sistema sarà suddiviso in due componenti principale, ovvero una parte hardware che costituisce il dispositivo wearable per la gestione dei sensori e una parte software per la comunicazione fra wearable device e applicazione mobile. La parte software andrà a definire una libreria da utilizzare nello sviluppo di applicazioni mobile. Inoltre il sistema dovrà avere determinati attributi: Innanzitutto tutti il sistema dovrà essere compatto e autosufficiente per garantire le caratteristiche di un dispositivo wearable, in seguito dovrà essere affidabile e procurare informazioni consistenti, infine dovrà garantire reattività alle richieste dell'utente.

2.1 Requisiti Funzionali

Nel seguente caso di studio vi è la necessità di utilizzare molteplici sensori gestiti da un dispositivo di dimensioni contenute (denominato server) che funge da unità computazionale. I valori saranno poi visualizzati dalle applicazioni sviluppate utilizzando la libreria proposta dalla piattaforma. Detto ciò, le componenti principali della piattaforma saranno un dispositivo wearable (server) a cui sono collegati diversi sensori e una libreria per lo sviluppo di applicazioni mobile, chiamate di seguito client.

Innanzitutto si analizzano i requisiti della componente server del sistema: per eviscerare tutte le caratteristiche che esso deve avere è necessario descrivere le problematiche a cui il server andrà incontro.

Il primo fattore da approfondire riguarda le caratteristiche dei sensori. I sensori utilizzati per il rilevano di parametri vitali possono essere suddivisi in base a due caratteristiche, ovvero funzionamento e tipologia di informazioni rilevate.

Considerato il funzionamento, alcuni sensori, come ad esempio il sensore della temperatura corporea, richiedono una interazione minima con l'utilizzato, ovvero applicarlo al paziente e collegarlo al dispositivo. Altri sensori più complessi, come il rilevatore della pressione, richiedono un certo numero di step da seguire per ottenere i valori desiderati.

Considerando invece la tipologia di informazioni prodotte dai sensori, alcuni di essi non richiedono un utilizzo continuo, ovvero l'informazione ottenuta è relativa e parametri non mutevoli nel breve periodo. Questi sensori non richiedono una continua rilevazione da parte dell'unità di controllo, la quale esegue letture del sensore in esame a frequenze molto basse. Nel caso più estremo l'unità di controllo esegue una singola lettura del sensore, aspettando eventuali richieste da parte dell'utente nel caso in cui sia effettivamente necessario un valore aggiornato. La seconda tipologia di sensori invece genera una mole di dati molto consistente per descrivere le informazioni di cui si occupa. Queste informazioni sono infatti trasformate in un flusso (o stream) di dati continuo che richiede molta attenzione da parte dell'unità computazionale.

Il server dovrà essere in grado di gestire le tipologie di sensori descritte in precedenza adattandosi alle circostanze in cui si trova. È importante che l'unità di controllo sappia rilevare i sensori a cui è collegato per evitare uno spreco di risorse. L'efficienza con cui il server gestisce i sensori permetterà al sistema di mantenere un certo livello di reattività alle richieste effettuate dall'utente.

Per quanto riguarda la libreria, essa dovrà rendere disponibili all'applicazione che la utilizza delle API per il controllo della comunicazione, la ricezione e l'invio di messaggi col server. Nello specifico, la libreria deve implementare un sistema di comunicazione locale fra il dispositivo su cui è in esecuzione l'applicazione e il sottosistema che gestisce i sensori. Le funzionalità da implementare sono la ricezione dei parametri vitali rilevati dai sensori e l'invio di richieste per singoli valori di interesse.

Riassumendo, le modalità di ricezione delle informazioni elaborate dal sistema si distinguono in due categorie:

- Messaggi push, in cui l'utente legge i parametri inviati dal il sistema in maniera singola o continua(stream),

- Modalità pull (o richiesta/risposta), in cui l'utente richiede attivamente un determinato valore di interesse.

Entrambe le modalità sono implementate dalla libreria. Per mantenere il controllo sul dominio dei messaggi, la libreria dovrà inoltre costruire un protocollo per la comunicazione adatto alle esigenze.

L'ultima annotazione da fare riguarda le applicazioni mobile: nello sviluppo di questa piattaforma è necessario tenere in considerazione il target a cui la piattaforma fa riferimento, ovvero le applicazioni. Tutte le scelte da effettuare dovranno tener conto di questo dettaglio.

2.2 Requisiti non Funzionali

Dato che le informazioni elaborate dal server sono visualizzabili dall'utente soltanto se vi è collegato un client, è possibile predisporre il server affinché entri in modalità standby nel momento in cui nessuno sia disponibile alla visualizzazione delle informazioni: questo dettaglio aumenta l'autonomia del sistema.

Un altro aspetto interessante che potrebbe aumentare le capacità del sistema riguarda l'accesso multiplo al server: si potrebbe predisporre il server per comunicare con molteplici client in contemporanea per permettere a più utenti di visualizzare le informazioni che cercano. Nel momento in cui più client si connettono, il server potrebbe cambiare configurazione, limitando lo scambio di informazioni di routine e concentrandosi sulle richieste effettuate dai differenti client. Ovviamente queste modifiche al client richiedono una modifica della libreria e soprattutto del protocollo di comunicazione.

Un altro aspetto da considerare riguarda la latenza del sistema a fronte di una richiesta dell'utente: è importante decidere quanto il sistema debba essere real-time nelle risposte, in base all'importanza dell'informazione richiesta e al carico di lavoro che l'unità computazionale sta sopportando al momento della richiesta.

Collegando questo concetto con la possibilità di avere più client collegati, sarà necessario decidere in che modo il server deve approcciarsi a richieste multiple che potrebbero provenire da differenti client. La libreria potrebbe impedire l'invio di richieste multiple oppure predisporre una coda di richieste per inviare singolarmente al server. In alternativa, questa coda potrebbe essere gestita direttamente dal server.

Infine, considerando la caratteristiche della sensoristica, potrebbe essere interessante offrire all'utente l'opportunità di configurare il sistema in base alle proprie preferenze. La libreria potrebbe predisporre delle API per la configurazione dei sensori, cambiandone per esempio la frequenza di campionamento.

2.3 Casi d'uso

Una volta analizzati tutti i requisiti ricavati dal caso di studio, è possibile descrivere una architettura generale del funzionamento del sistema. Il diagramma di seguito mostra le interazioni che l'utente può avere col sistema:

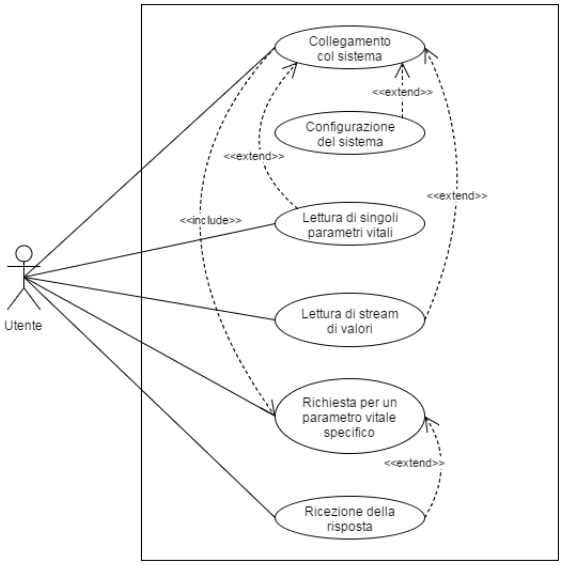
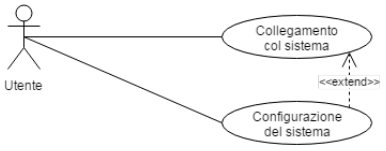


Figura 2.1: Casi d'uso del sistema

2.3.1 Connessione e configurazione del sistema

Per poter visualizzare i parametri vitali ricavati dai sensori, è necessario eseguire un collegamento fra applicazione e sistema. La libreria fornisce delle API per la connessione col server e per il controllo del suo stato. Una volta eseguita una connessione si potrebbero specificare alcune preferenze utilizzando API dedicate alla configurazione del sistema.



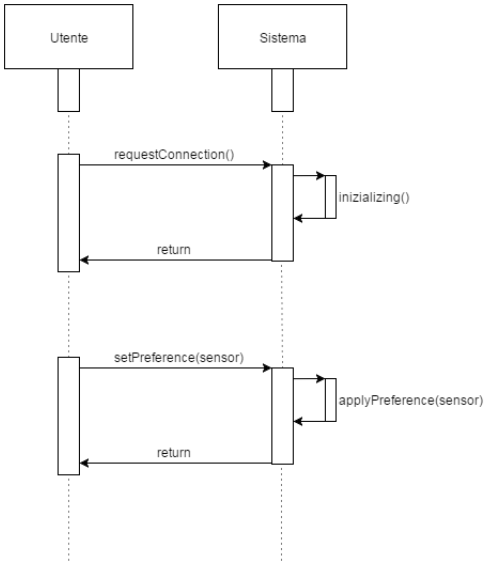
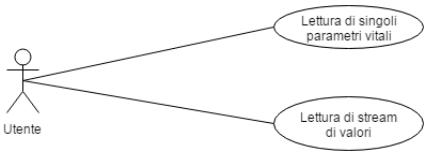


Figura 2.2: Diagramma di sequenza relativo alla connessione col sistema e configurazione

2.3.2 Ricezione dei messaggi (modalità push)

Nel momento in cui la connessione è stata stabilità, il sistema inizia a rilevare i dati prodotti dai sensori e a inviarli all'utente. La libreria raccoglie e cataloga tutti i messaggi ricevuti dal sistema ed offre all'applicazione delle API per poterli leggere. In questo modo l'utente può leggere sull'applicazione tutti i parametri vitali rilevati in quel momento.



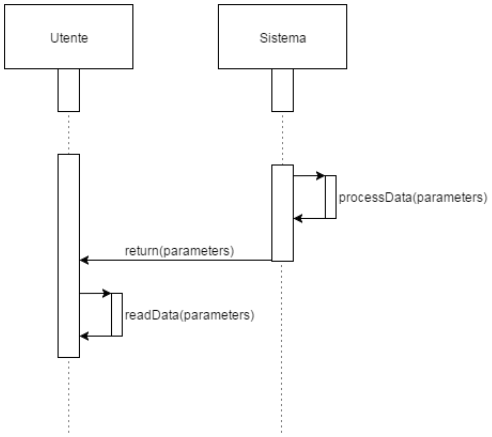


Figura 2.3: Diagramma di sequenza relativo alla lettura dei parametri vitali

2.3.3 Richiesta di un valore (modalità pull)

La libreria offre delle API per effettuare richieste per determinati valori di interesse. Una volta che il sistema ha ricevuto la richiesta, esso esegue una lettura real-time del valore desiderato dall'utente ed invia un messaggio all'applicazione il prima possibile. La libreria cattura la risposta del sistema ed offre una API per la lettura del dato appena ricevuto da parte dell'applicazione.

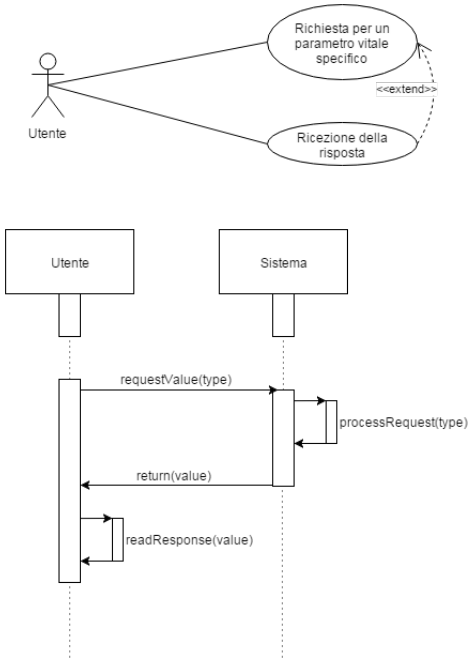


Figura 2.4: Diagramma di sequenza relativo alla modalità richiesta/risposta

Capitolo 3

Modellazione e Tecnologie utilizzate

Nella prima parte di questo capitolo viene modellato il dominio applicativo del sistema. In seguito vengono mostrate tutte le scelte tecnologiche effettuate. Per comodità verranno utilizzate le parole chiave chiave client e server per indicare rispettivamente l'applicazione mobile e la piattaforma (nello specifico la componente hardware).

3.1 Dominio applicativo dei messaggi

Di seguito sono elencati i tutti i tipi di messaggi scambiati fra client e server, descrivendo il tipo di informazioni invitata. Da notare che l'inizializzazione non è accennata in quanto essa è utilizzata solamente per creare una connessione locale stabile fra client e server e nessuna informazione inerente ai requisiti viene scambiata.

3.1.1 Invio valori singoli (da Server a Client)

Nel momento in cui un client è connesso al server ed esso rileva la presenza di un sensore che produce singoli valori significativi, come descritto nei requisiti funzionali, il server invia un singolo messaggio al client contenente le seguenti informazioni:

- ID sensore,
- Valore intero o decimale che descrive l'informazione rilevata dal sensore.

3.1.2 Invio stream di valori (da S a C)

Nel momento in cui un client è connesso al server ed esso rileva la presenza di un sensore complesso che genera un flusso di dati costante, il server inizia a trasmettere in loop le informazioni generate dal sensore. Ogni singolo messaggio utilizzato nello stream ha la seguente struttura:

- ID sensore,
- ID stream,
- Valore intero o decimale.

Si noti che l'ID stream è necessario solo nel caso in cui il sensore in esame necessiti di più stream di valori per comunicazione le informazioni che rileva.

3.1.3 Invio richiesta per una informazione (da C a S)

Quando un utente vuole ottenere una certa informazione, utilizza il client per inviare una richiesta al server. La richiesta è un messaggio molto semplice contenente una singola stringa che descrive l'informazione che cui si ha bisogno.

3.1.4 Invio richiesta per una informazione (da C a S)

Quando il server riceve una richiesta dal client, esso effettua una lettura del sensore responsabile per dell'informazione e risponde al client con un messaggio contenente le seguenti informazioni:

- Tipologia di messaggio (risposta),
- ID sensore,
- Valore intero o decimale che descrive l'informazione rilevata dal sensore.

Si noti che la tipologie di messaggio è indispensabile per distinguere questo messaggio dalle informazioni inviate coi messaggi singoli.

3.2 Scelta delle tecnologie

Come già accennato in precedenza, la scelta delle tecnologie da adottare è cruciale per lo sviluppo di questa piattaforma. Scelte differenti comportano cambiamenti significativi al funzionamento del sistema. Come già accennato in precedenza, le problematiche principali riguardano:

- L'unità di controllo (Server),

- Il sistema di comunicazione.

Prima di approfondire questi problemi, è necessario specificare fin dal principio che la scelta del sistema operativo su cui è basata la piattaforma (Android, iOS, ecc.) va a influenzare in maniera minima (se non nulla) le scelte tecnologiche, progettuali ed implementative effettuate nello sviluppo della piattaforma stessa.

3.3 Unità di controllo

Una componente che richiede molta attenzione è l'unità di controllo per la gestione della sensoristica. La prima decisione che è stata presa riguarda la tipologia di unità di calcolo da utilizzare. Considerando le caratteristiche wearable che bisogna rispettare e la mole relativamente contenuta di informazioni che si deve gestire, la decisione è ricaduta sui single-board computer (già accennati nello stato dell'arte), molto in voga in questi ultimi anni e facilmente accessibili. Questa scelta va a influenzare tutte le scelte che verranno fatto in seguito. Detto ciò è necessario analizzare la struttura logica che il dispositivo deve avere per svolgere al meglio i compiti prestabiliti.

Come già enunciato, i sensori si possono suddividere in due tipologie, ma ognuno di essi necessita di un trattamento specifico per la rilevazione dei dati: alcuni sensori utilizzano protocolli di comunicazione conosciuti come I2C, mentre altri sono più semplici e immediati. Risulta quindi molto comodo l'utilizzo di una scheda aggiuntiva e di una libreria per l'interfacciamento coi sensori.

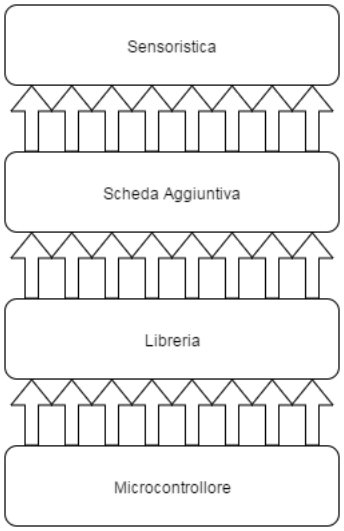


Figura 3.1: Livelli logici del Server

La struttura di base del sistema risulta composta da:

- Un assortimento di sensori per il rilevamento dei parametri vitali,
- Una scheda aggiuntiva per il collegamento fisico dei sensori dal sistema,
- Una libreria per l'interfacciamento coi sensori,
- Un single-board computer che utilizzare la libreria per eseguire le operazioni di routine.

La scelta dei componenti si è divisa in due fasi: scelta della scheda aggiuntiva e scelta del mini-computer.

Per quanto riguarda la scheda aggiuntiva, la scelta è ricaduta sulla scheda e-Health Sensor Platform v2.0 prodotta dalla Cooking Hacks e accennata nello stato dell'arte. Nonostante la completezza di questo pacchetto vi sono alcune limitazioni che riguardano sia il funzionamento della scheda stessa sia alcune decisioni future. In primis, è importante specificare che questa scheda è stato sviluppata per essere utilizzata in accoppiata con uno fra due single-computer board, ovvero Arduino o Raspberry Pi (quest'ultimo con l'ausilio di un adattatore). Questa limitazione non risulta essere un problema sostanziale, in quanto entrambe sono già ampiamente utilizzate nel mondo dei sistemi embedded e wearable con grande successo. L'altro aspetto negativo che invece causa problemi più importanti riguarda il funzionamento stesso della scheda e-Health. Dopo una analisi della libreria e alcuni test, si è notato che alcuni sensori utilizzano le stesse aree di memoria per lo scambio di informazioni, e ciò

può causare letture inconsistenti. Nonostante queste limitazione, il prodotto risulta comunque valido e consono allo sviluppo di questo progetto.

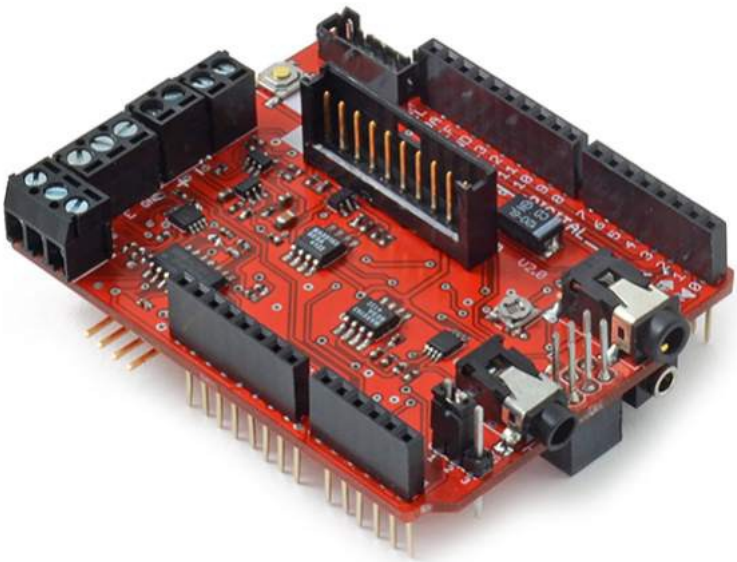


Figura 3.2: Scheda e-Health

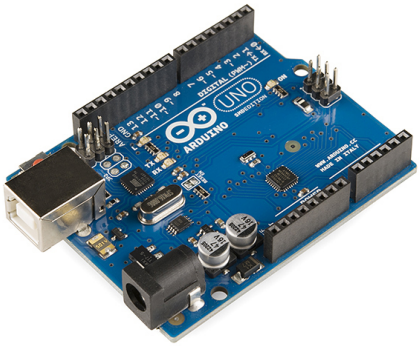
Per quanto riguarda la scelta del single-board computer, le opzioni sono vincolate fra Arduino e Raspberry Pi. Da una parte si ha un'unità computazionale molto semplice basata su microcontrollore, dall'altra parte si ha una scheda con microprocessore e sistema operativo.

In una architettura a microcontrollore (arduino nel caso in esame) non vi è il supporto di un sistema operativo, quindi non è possibile effettuare alcune operazioni utili come il multi-tasking. Come già accennato nello stato dell'arte, una volta alimentato, il calcolatore esegue l'unico programma caricato in memoria in un ciclo infinito, ovvero un metodo loop eseguito ciclicamente. Il supporto per la programmazione di Arduino è il framework chiamato Wiring, che offre un linguaggio che eredità moltissime caratteristiche dal linguaggio C/C++.

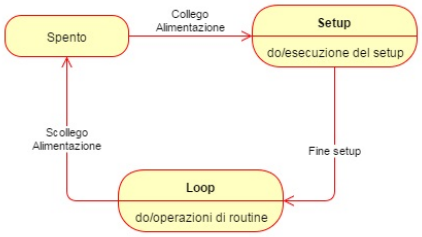
Utilizzando Raspberry Pi si ha a disposizione un maggiore capacità di calcolo e il supporto di un sistema operativo, quindi è possibile utilizzare linguaggi

di alto livello come java per sviluppare programmi strutturalmente molto simili a quelli sviluppati su un normale computer general-purpose.

La scelta è ricaduta su Arduino principalmente per comodità, data la maggiore familiarità con quest'ultimo rispetto a Raspberry.



(a) Scheda Arduino uno



(b) Diagramma a stati sul funzionamento

Figura 3.3: Unità di controllo (Server)

Utilizzando Arduino non vi è la possibilità di tener traccia dello stato della connessione, quindi non sarà possibile utilizzare la modalità standby. Questa problematica sarà affrontata nei prossimi capitoli.

3.4 Comunicazione

Dato che il target della piattaforma è la programmazione mobile, le soluzioni più comode adottabili sono una connessione bluetooth o una connessione wifi.

In entrambi i casi il sistema si comporta come segue:

- Lato Client: sono disponibili librerie standard per la gestione della connessione con entrambe le tecnologie,
- Lato Server: si necessità dell'utilizzo di un modulo(componente) esterno.

La tecnologia adottata è stata bluetooth per via della maggior familiarità con essa.

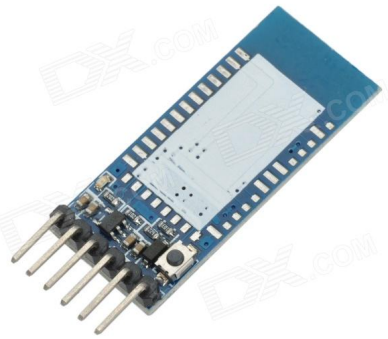


Figura 3.4: Modulo bluetooth ZS-040

Questo modulo funge esclusivamente da slave, quindi può solo accettare richieste di connessione e gestisce la socket di comunicazione in maniera automatica. Per inviare e riceve dati, Arduino utilizza una libreria standard chiamata SoftwareSerial. La scelta di questo modulo impedisce la gestione di connessione multi-client.

Capitolo 4

Progettazione

4.1 Architettura logica generale del sistema

Il sistema è basato su una architettura che ricorda molto quella Client-Server, in cui il server è composto dall'unità di controllo della sensoristica, quindi Arduino, scheda e-Health e sensori, mentre il client è l'applicazione su smartphone che si occupa della visualizzazione delle informazioni.

L'architettura generale è quindi la seguente:

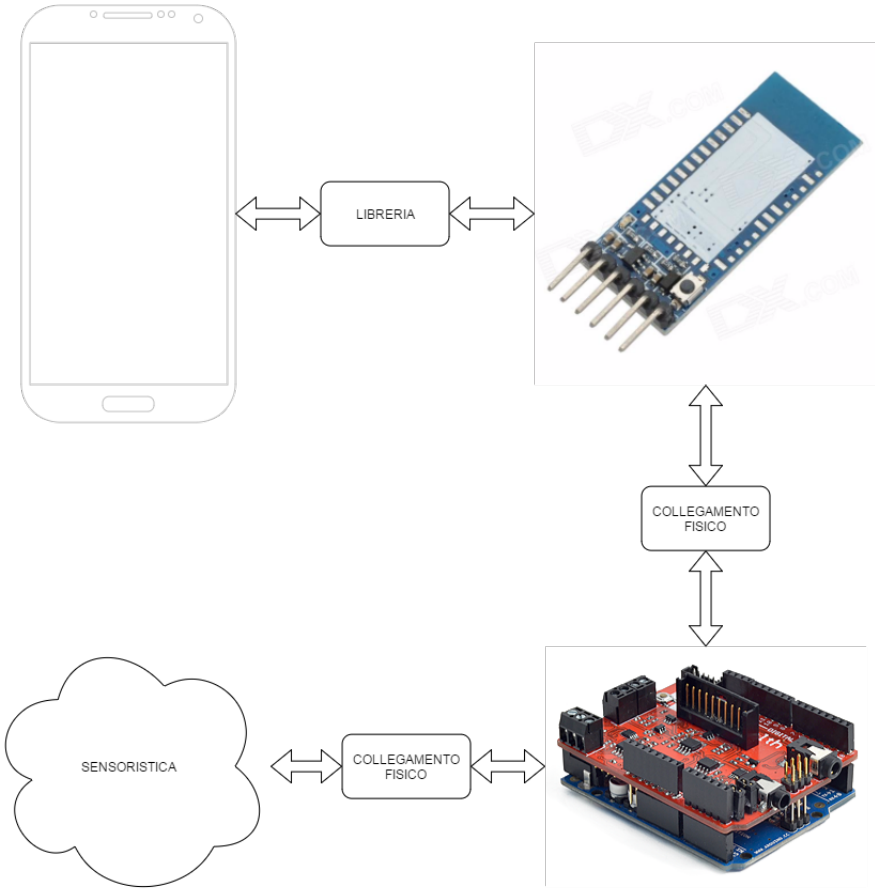


Figura 4.1: Architettura generale del sistema

Si avrà quindi una applicazione in esecuzione su smartphone che effettua una richiesta di connessione ad arduino attraverso il bluetooth. Nel momento in cui la connessione è stata stabilità, Arduino invia le informazioni ottenute dai sensori allo smartphone utilizzando il modulo bluetooth a cui è collegato. è importante includere l'applicazione nell'architettura generale anche se non fa parte strettamente del progetto, in quanto aiuta a comprendere al meglio il funzionamento finale della piattaforma.

4.2 Suddivisione del progetto

Basandosi sulle caratteristiche funzionali del sistema e sulle scelte tecnologiche effettuate in precedenza, considerando le limitazioni che sono emerse, il progetto è stato suddiviso in 3 parti:

- Libreria per la comunicazione,

- Applicazione di prova,
- Software Arduino.

Di queste 3 parti, la libreria richiederà uno studio approfondito per risolvere i problemi incontrati fino ad ora, introducendo alcune funzionalità aggiuntive non specificate nei requisiti. Inoltre vi sarà un breve paragrafo riguardante l'architettura generale del modulo Arduino. Infine, non è necessario introdurre alcuna struttura dell'applicazione, in quanto qualunque applicazione che implementa la libreria rispetterà i requisiti richiesti.

4.3 Libreria

La comunicazione fra Arduino e Smartphone, come già anticipato, è effettuata con bluetooth.

Tutto il sistema deve adattarsi ai limiti computazionali di Arduino, quindi le comunicazione devono essere ridotte al minimo e soprattutto devono contenere il numero minimo di informazioni. Il compito di questa libreria è quello di proporre un protocollo di comunicazione affidabile ed essenziale affinché Arduino sia strutturato al meglio, mantenendo un certo livello di modularità, permettendo modifiche future.

La libreria è suddivisa su due livelli di astrazione:

- Livello di comunicazione,
- Livello di interfaccia utente.

Ognuno di questi propone delle API ai livelli superiori.

Ovviamente la libreria riguarda soltanto il client, ovvero su smartphone, in quanto Arduino non permette di costruire un sistema di comunicazione avanzato. Arduino non ha alcun modo di conoscere lo stato della connessione.

4.3.1 Protocollo di comunicazione e problematiche

Prima di analizzare l'architettura della libreria, è necessario introdurre il protocollo dei messaggi accennato in precedenza. Esso definisce la struttura e il tipo di messaggi da utilizzare durante la comunicazione fra Arduino e Smartphone. Questo protocollo è necessario per distinguere i messaggi fra loro e snellire le comunicazione il più possibile.

La maggior parte dei messaggi viene inviata da Arduino verso Smartphone e sono divisi in due categorie:

- Messaggi di routine, ovvero i parametri vitali rilevati dai sensori,

- Risposte alle richieste, ovvero messaggi particolari inviati allo smartphone come risposta ad una determinata richiesta.

I messaggi inviati da smartphone verso Arduino sono quelli relativi alle richieste dell'utente. Questi messaggi sono molto minimali per facilitare Arduino nell'operazione di lettura di questi messaggi.

È necessario analizzare le problematiche legate alla perdita di connessione. Per quanto riguarda i messaggi di routine, la disconnessione non risulta essere un problema, in quanto queste informazioni vengono inviate ciclicamente a frequenza constante. Al momento della riconnessione, i dati persi sono già obsoleti, quindi non vi è la necessità di costruire un sistema di recupero dei dati. Per quanto riguarda le richieste, la libreria predisporrà un sistema in grado di verificare se la richiesta è stata smarrita e quindi notificare all'utente dell'errore avvenuto.

4.3.2 Architettura generale della libreria

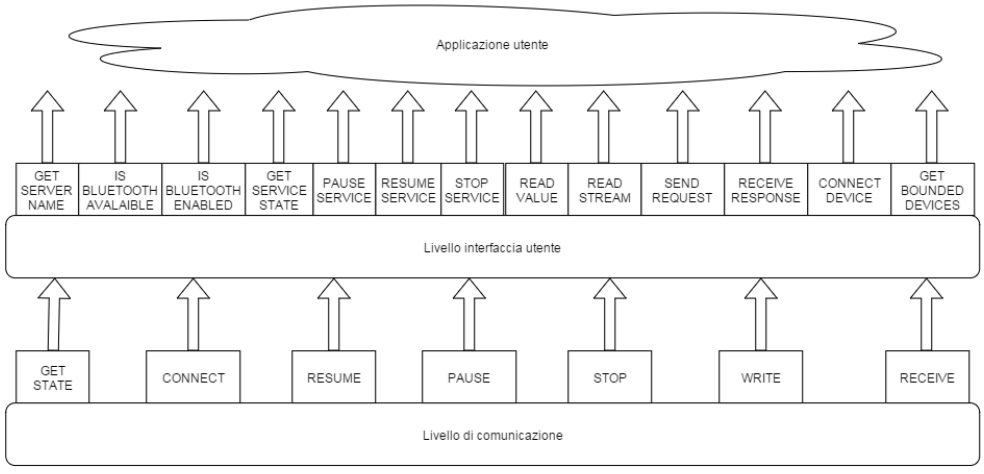


Figura 4.2: Architettura generale della libreria

4.3.3 API fornite dal livello interfaccia utente

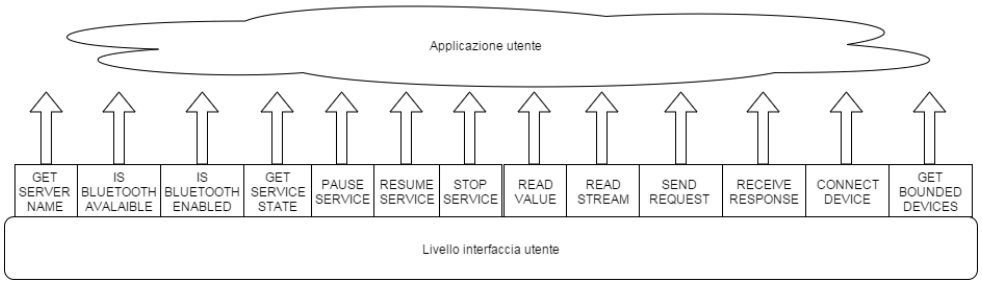


Figura 4.3: API fornite dal livelllo interfaccia utente

GET SERVER NAME: restituisce al client il nome del modulo bluetooth utilizzato da arduino per la comunicazione.

IS BLUETOOTH AVAILABLE: verifica se lo smartphone possiede il bluetooth.

IS BLUETOOTH ENABLED: verifica se il bluetooth è attivo sullo smartphone.

GET SERVICE STATE: restituisce lo stato attuale della connessione.

PAUSE SERVICE: mette la connessione in pausa.

RESUME SERVICE: esce dalla pause e riattiva la connessione.

STOP SERVICE: chiude la connessione col server.

READ VALUE: permette di leggere dalla libreria un parametro inviato dal server.

READ STREAM: permette di leggere dalla libreria uno stream di valori inviati dal server.

SEND REQUEST: invia una richiesta al server per un determinato valore di interesse.

RECEIVE RESPONDE: riceve la risposta dal server.

CONNECT DEVICE: permette allo smartphone di collegarsi al modulo Arduino.

GET DOUNDED DEVICES: restituisce all'utente i server a cui è possibile collegarsi.

4.3.4 Composizione del livello di comunicazione

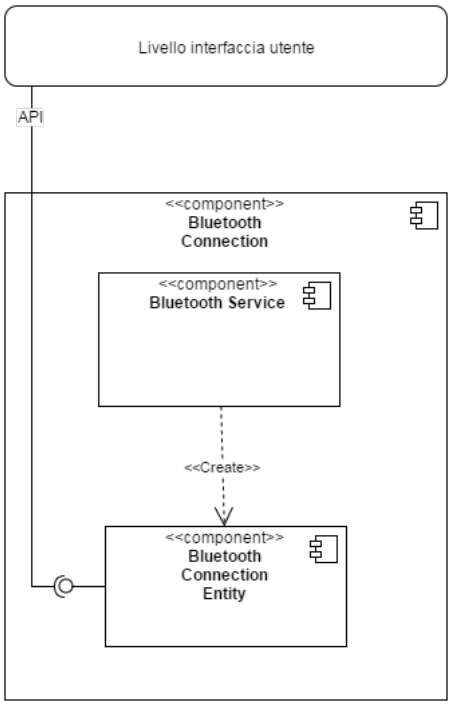


Figura 4.4: Architettura del livello di comunicazione

La componente Bluetooth Service viene creata dal livello di interfaccia utente quando viene inizializzata la libreria. Nel momento in cui viene instaurata una connessione col server, il Service crea una seconda entità Bluetooth Connection Entity che si occupa di mantenere attiva la connessione fra client e server. Inoltre questa entità fornisce le API per il livello interfaccia utente.

In seguito è mostrato il diagramma di sequenza per l'inizializzazione della connessione e la ricezione di un messaggio inviato dal server.

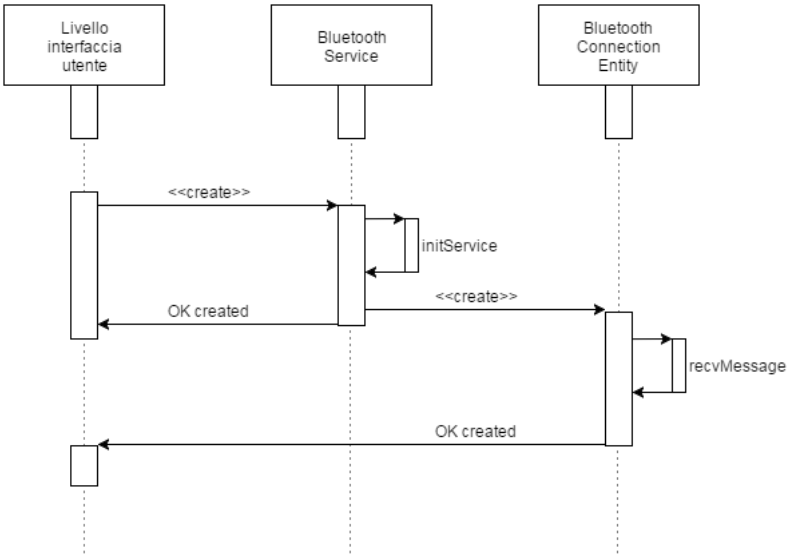


Figura 4.5: Diagramma di sequenza per inizializzazione della connessione e ricezione di un messaggio inviato dal server

In seguito è descritto lo scenario dell'invio di un messagio al server:

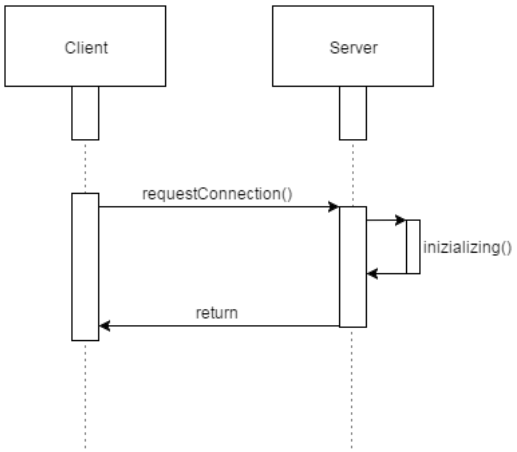


Figura 4.6: Diagramma di sequenza per l'invio di un messaggio al server

4.4 Modulo Arduino

4.4.1 Problematiche

Come già anticipato, per determinare l'architettura del software Arduino è necessario tenere in considerazione alcuni fattori:

- Sensoristica,
- Limitazioni computazionali del microcontrollore,
- Reattività,
- Perdita della connessione.

La sensoristica presenta diverse tipologie di sensori da trattare in maniera differente.

Le limitazioni computazionali richiedono un utilizzo ottimale delle risorse e scelte implementative efficienti.

La reattività è un requisito da rispettare entro i limiti della tecnologia utilizzata.

Infine, per quanto riguarda la perdita di connessione, la soluzione è implementata nella libreria, in quanto Arduino non è in grado di tener traccia dello stato della connessione.

4.4.2 Soluzioni adottate

L'idea di base è quella di modularizzare la gestione dei sensori, predisponendo un modulo per ogni sensori. Essi saranno poi gestiti dalla classe principale, che si occuperà anche delle richieste inviate dall'utente.

L'architettura generale è quindi la seguente:

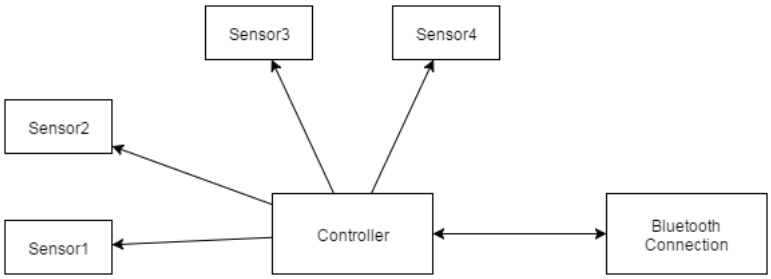


Figura 4.7: Architettura generale del Server

Ogni modulo sarà indipendente dagli altri e, a seconda dal tipo di sensore di cui si occupa, lavorerà in maniera differente. Per alleggerire il carico computazione ed adattarsi alle caratteristiche di ogni sensore, ogni modulo avrà una frequenza di rilevamento dei dati differente.

Per quanto riguarda la gestione delle richieste, il microcontrollore non riceve alcun tipo di segnalazione al momento della ricezione di un messaggio. Per controllare l'arrivo di richieste sarà necessario eseguire attivamente un controllo

sul modulo bluetooth, a una frequenza sufficientemente alta da non smarrire le richieste.

In seguito è descritta la procedura di ricezione e risposta ad una richiesta inviata dal client:

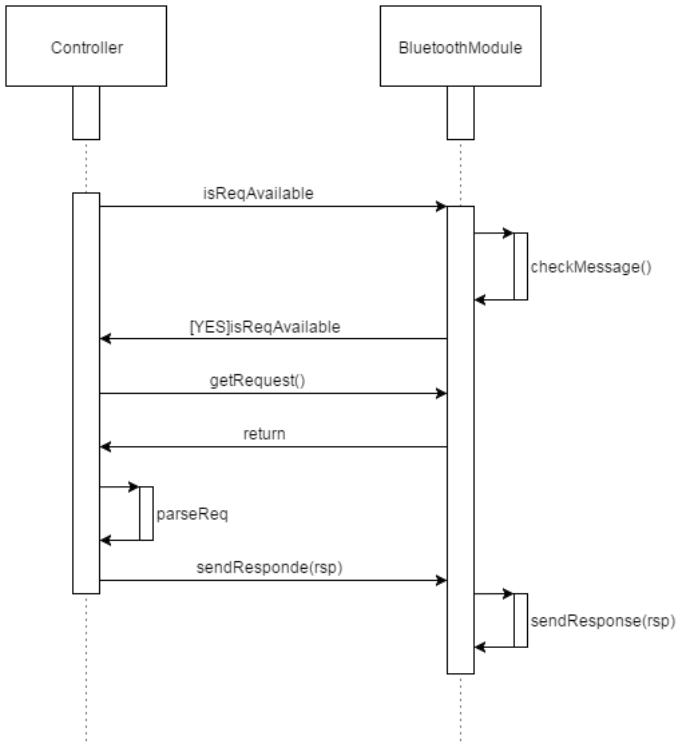


Figura 4.8: Diagramma di sequenza del meccanismo richieste/risposta

Capitolo 5

Prototipo

In questo capitolo vengono descritte le scelte implementative adottate nello sviluppo del progetto. Prima di proseguire è necessario specificare che alcune delle funzionalità proposte nei requisiti non funzionali non sono state implementate: il sistema non gestisce connessioni multi-client e non prevede la gestione di richieste multiple. Inoltre non è possibile effettuare operazioni di configurazione.

Un'ultima considerazione da fare riguarda la scelta del sistema operativo utilizzato per l'implementazione della piattaforma: in questo prototipo è stato utilizzato Android come linguaggio per lo sviluppo. Da notare che questa scelta non influenza le decisioni prese in fase di progettazione.

5.1 Implementazione della libreria

La libreria è strutturata in 3 classi:

- Constants,
- BluetoothService: implementata il livello di connessione,
- BluetoothSPP: implementa il livello interfaccia utente.

La classe **Constants** è utilizzata per mantenere il controllo della libreria. Essa definisce lo stato della connessione e il protocollo di comunicazione.

La classe **BluetoothService** si occupa della connessione bluetooth fra Applicazione Android e Arduino. Per non imporre carico di lavoro aggiuntivo al thread principale, la connessione è gestita da 2 thread ognuno dei quali si occupa di una parte specifica della connessione.

```
private class ConnectThread extends Thread {
       private final BluetoothSocket mmSocket;
       private final BluetoothDevice mmDevice;

       public ConnectThread(BluetoothDevice device) {
           mmDevice = device;
           BluetoothSocket tmp = null;
           try {
               tmp = device.createRfcommSocketToServiceRecord(
                       MY_UUID);
           } catch (IOException e) { }
           mmSocket = tmp;
       }

       public void run() {
           setName("ConnectThread");
           mAdapter.cancelDiscovery();
           try {
               mmSocket.connect();
           } catch (IOException e) {
               try {
                   mmSocket.close();
               } catch (IOException e2) { }
               connectionFailed();
               return;
           }
           synchronized (BluetoothService.this) {
               mConnectThread = null;
           }
           connected(mmSocket, mmDevice);
       }

       public void cancel() {
           try {
               mmSocket.close();
           } catch (IOException e) { }
       }
   }


   private class ConnectedThread extends Thread {
       private final BluetoothSocket mmSocket;
       private final InputStream mmInStream;
```

```
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {
        mmSocket = socket;
        InputStream tmpIn = null;
        OutputStream tmpOut = null;
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) { }
        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }

    public void run() {
        byte[] buffer;
        ArrayList<Integer> arr_byte = new ArrayList<Integer>();
        while (true) {
            try {
                int data = mmInStream.read();
                if(data == 0x0A) {
                } else if(data == 0x0D) {
                    buffer = new byte[arr_byte.size()];
                    for(int i = 0 ; i < arr_byte.size() ; i++) {
                        buffer[i] = arr_byte.get(i).byteValue();
                    }
                    mHandler.obtainMessage(Constants.MESSAGE_READ
                            , buffer.length, -1,
                                buffer).sendToTarget();
                    arr_byte = new ArrayList<Integer>();
                } else {
                    arr_byte.add(data);
                }
            } catch (IOException e) {
                if(currentConnectedDevice != null)
                    connectionLost();
                break;
            }
        }
    }

    public void write(byte[] buffer) {
        try {
            mmOutStream.write(buffer);
```

```
                mHandler.obtainMessage(Constants.MESSAGE_WRITE, -1,
                    -1, buffer)
                        .sendToTarget();
            } catch (IOException e) { }
        }

        public void cancel() {
            try {
                mmSocket.close();
            } catch (IOException e) { }
        }
    }
```

ConnectThread esegue la connessione con un modulo Arduino nel momento in cui l'utente lo desidera. ConnectedThread mantiene la socket aperta per l'invio e la ricezione di messaggi.

BluetoothService offre al livello superiore delle API per la gestione dello stato della connessione e per l'invio di richieste al server

```
public synchronized int getState() {
    return mState;
}

public void write(byte[] out) {
    ConnectedThread r;
    synchronized (this) {
        if (mState != Constants.STATE_CONNECTED) return;
        r = mConnectedThread;
    }
    r.write(out);
}
```

Per la ricezione dei messaggi, la classe predispone un handler che sarà poi implementato dal livello superiore. Questo handler, oltre a contenere i messaggi ricevuti dal server, gestisce tutti i messaggi di servizio utilizzati dalla libreria. Ecco un esempio di messaggio di servizio:

```
    Message msg = mHandler.obtainMessage(Constants.MESSAGE_TOAST);
    Bundle bundle = new Bundle();
    bundle.putString(Constants.TOAST, "Device connection was lost");
    msg.setData(bundle);
    mHandler.sendMessage(msg);
```

Infine, BluetoothService gestisce la perdita di connessione con arduino, fornendo allo strato superiore delle API per il salvataggio e il ripristino della connessione. Il sistema utilizzato per la riconnessione è molto semplice: nel momento in cui la connessione viene persa, la libreria tenta di creare una nuova connessione con l'ultimo Arduino a cui era collegato un numero prefissato di volte. Se la connessione è ristabilita lo scambio di messaggi viene ripristinato, altrimenti l'utente deve attivamente effettuare una nuova connessione utilizzando le API del livello interfaccia utente.

La classe **BluetoothSPP** propone all'utente tutte le API necessarie al corretto utilizzo del Bluetooth. Questa classe è statica per renderla il più riusabile possibile. In seguito viene presentato l'handler accennato nella classe precedente:

```
private static final Handler handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case Constants.MESSAGE_STATE_CHANGE:
                if(mBluetoothStateListener != null)
                    mBluetoothStateListener.
               onServiceStateChanged(msg.arg1);
                break;
            case Constants.MESSAGE_WRITE:
                Toast.makeText(context,"Wait for
                    Response",Toast.LENGTH_SHORT).show();
                break;
            case Constants.MESSAGE_READ:
                byte[] readBuf = (byte[]) msg.obj;
                String readMessage = new String(readBuf, 0, msg.arg1);
                if(readBuf != null && readBuf.length > 0) {
                    String[] vett = readMessage.split(":");
               String temp = "";
                    switch(vett[0]) {
                        case Constants.BREATH_VALUE:
                                if (mBreathDataReceivedListener != null)
                                    mBreathDataReceivedListener.
                           onDataReceived(vett[1]);
                                break;
                   case Constants.TEMPERATURE_VALUE:
                                if (mTemperatureDataReceivedListener !=
                                    null)
                                    temp = Double.parseDouble(vett[1])
                                        + " \u00B0C";
```

```
                                mTemperatureDataReceivedListener.
                        onDataReceived(temp);
                            break;

//vi e' un case per ogni valore
                        case Constants.RESPONSE_TEMPERATURE:
                                if (mResponseDataReceivedListener !=
                                    null) {
                                    temp = Double.parseDouble(vett[1])
                                        + " \u00B0C";
                                    mResponseDataReceivedListener.
                        onTemperatureDataReceived(temp);
                                    isRequestHandled = true;
                                    expireTimer.cancel();
                                }
                                break;
                    case Constants.RESPONSE_PULSE:
                       if (mResponseDataReceivedListener != null) {
                          temp = Double.parseDouble(vett[1]) + " bpm";
                          mResponseDataReceivedListener.
                          onPulseDataReceived(temp);
                          isRequestHandled = true;
                          expireTimer.cancel();
                       }
                       break;
//vi e' un case per ogni risposta
                        }
                }
                break;
            case Constants.MESSAGE_DEVICE_NAME:
                mConnectedDeviceName =
                    msg.getData().getString(Constants.DEVICE_NAME);
                Toast.makeText(context, "Connected to " +
                    mConnectedDeviceName, Toast.LENGTH_SHORT).show();
                break;
            case Constants.MESSAGE_TOAST:
                Toast.makeText(context,
                    msg.getData().getString(Constants.TOAST),
                        Toast.LENGTH_SHORT).show();
                break;
        }
    }
};
```

Questa classe propone un protocollo di comunicazione leggero ed essenziale per la comunicazione fra Android e Arduino. I listener che si possono vedere in questa classe sono utilizzati per notificare all'applicazione che è stato ricevuto un parametro vitale di un certo tipo. A livello implementativo le informazioni relative a valori singoli e a stream di valori vengono trattate allo stesso modo: l'applicazione ha il compito di fornire una interfaccia grafica idonea al tipo di valori che legge. Ecco alcuni esempi di listener utilizzati:

```
public interface OnPositionDataReceivedListener {
    public void onDataReceived(String message);
}

public static void
    setOnPositionDataReceivedListener(OnPositionDataReceivedListener
    listener) {
    mPositionDataReceivedListener = listener;
}

public interface OnResponseDataReceivedListener {
    public void onTemperatureDataReceived(String message);
    public void onPulseDataReceived(String message);
    public void onOxygenDataReceived(String message);
    public void onConductivityDataReceived(String message);
    public void onResistanceDataReceived(String message);
    public void onMuscleDataReceived(String message);
    public void onSystolicDataReceived(String message);
    public void onDiastolicDataReceived(String message);
    public void onGlucometerDataReceived(String message);
    public void onNoDataReceived();
}
public interface BluetoothStateListener {
    public void onServiceStateChanged(int state);
}

public static void setBluetoothStateListener (BluetoothStateListener
    listener) {
    mBluetoothStateListener = listener;
}
```

L'handler smista le informazioni fra i vari listener, mentre l'applicazione implementa i metodi proposti nel listener per ricavare i dati dalla libreria. Per il sistema delle richieste, BluetoothSPP offre dei metodi che incapsulano il protocollo di comunicazione utilizzato. Ecco in seguito alcuni esempi:

```
private static boolean send(String data) {
    if(btService.getState() == Constants.STATE_CONNECTED){
        if(isRequestHandled) {
            data += "\r\n";
            btService.write(data.getBytes());
            isRequestHandled = false;
            if(expireTimer != null)
                expireTimer.cancel();
            expireTimer = new CountDownTimer(3000, 1000) {
                @Override
                public void onTick(long millisUntilFinished) {
                    //do nothing
                }

                @Override
                public void onFinish() {
                    isRequestHandled = true;
                    if (mResponseDataReceivedListener != null) {
                        mResponseDataReceivedListener.
                  onNoDataReceived();
                    }
                }
            }.start();
        }
        return true;
    } else
        return false;
}

public static boolean requestTemperature(){
   return send(Constants.REQUEST_TEMPERATURE);
}

public static boolean requestPulse(){
   return send(Constants.REQUEST_PULSE);
}
```

Nel momento in cui viene inviata una richiesta con successo parte un timer, al termine del quale si presuppone che Arduino abbia perso la richiesta, quindi si notifica all'utente l'errore. Se Arduino riesce a rispondere in tempo, il timer viene bloccato dall'handler e nessun errore viene notificato all'applicazione. Può essere inviata una sola richiesta alla volta, in quanto non è stato implementato un sistema di coda di richieste. La libreria blocca il sistema di

invio di richieste nel momento in cui il timer sopracitato viene lanciato e lo sblocca quando viene rilevata una risposta o quando il timer termina.

Infine, BluetoothSPP offre una API per ottenere i Paired Device a cui connettersi:

```
public static Set<BluetoothDevice> getBondedDevices(){
    return btAdapter.getBondedDevices();
}
```

5.2 Implementazione dell'applicazione di prova

L'applicazione sviluppata offre una interfaccia a scorrimento orizzontale composta da tab. Vi sono in totale 5 tab, di cui 4 utilizzato per la visualizzazione dei dati e uno predisposto all'invio di richieste.

L'idea di base è quella di utilizzare una singola activity che gestisce i tab dell'interfaccia, i quali sono definiti in 5 fragment. Inoltre si predispone un'altra activity per la selezione dei bounded device. Di seguito è mostrato il codice relativo all'activity principale:

```
package com.example.android.stabile;

import android.content.pm.ActivityInfo;
import android.os.Bundle;
import android.support.design.widget.TabLayout;
import android.support.v4.app.FragmentActivity;
import android.support.v4.view.ViewPager;

import com.example.android.mylibrary.BluetoothSPP;


public class MainActivity extends FragmentActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        setRequestedOrientation(ActivityInfo.
      SCREEN_ORIENTATION_PORTRAIT);
        ViewPager viewPager = (ViewPager)
            findViewById(R.id.viewpager);
```

```
        viewPager.setAdapter(new
            SampleFragmentPagerAdapter(getSupportFragmentManager(),
                MainActivity.this));
        TabLayout tabLayout = (TabLayout)
            findViewById(R.id.sliding_tabs);
        tabLayout.setupWithViewPager(viewPager);
    }
//forse sto pezzo di codice e' da sistemare
    @Override
    protected void onStop() {
        super.onStop();
        BluetoothSPP.stopService();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        BluetoothSPP.stopService();
    }
}
```

L'oggetto tabLayout gestisce la visualizzazione dei tab mentre l'oggetto viewPager gestisce lo stile dei tab e lo scorrimento orizzontale. Al viewPager è passato come parametro una istanza di SampleFragmentPagerAdapter che definisce la struttura dei tab. Ecco il codice di questa classe:

```
package com.example.android.stabile;

import android.content.Context;
import android.support.v4.app.FragmentPagerAdapter;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.Fragment;

public class SampleFragmentPagerAdapter extends FragmentPagerAdapter
    {
    final int PAGE_COUNT = 5;
    private String tabTitles[] = new String[] { "Chart",
        "CurrentData",
            "Test", "Movement", "SendCmd" };
    private Context context;

    public SampleFragmentPagerAdapter(FragmentManager fm, Context
        context) {
        super(fm);
```

```
        this.context = context;
    }

    @Override
    public int getCount() {
        return PAGE_COUNT;
    }

    @Override
    public Fragment getItem(int position) {
        return PageFragment.newInstance(position + 1);
    }

    @Override
    public CharSequence getPageTitle(int position) {
        return tabTitles[position];
    }
}
```

Nell'interfaccia sono attivi 3 fragment alla volta, ovvero quello visibile dall'utente e quelli immediatamente a sinistra e a destra. Nel momento in cui si esegue uno scorrimento verso destra la pagine più a sinistra viene distrutta mentre a destra viene istanziato un nuovo fragment.

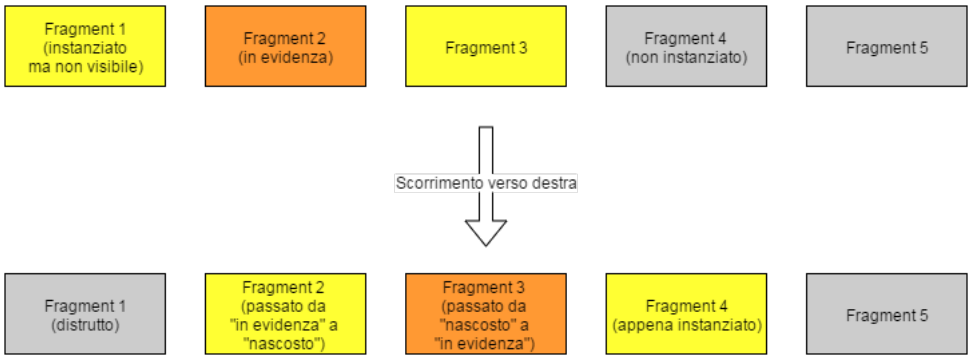


Figura 5.1: Funzionamento dei fragment

La classe PageFragment utilizzata nel metodo getItem del codice precedente descrive i componenti contenuti all'interno di ogni tab. Questa classe viene instanziata più volte, quindi si è adottato un ID per distinguere una istanza dall'altra. Di seguito sono mostrati alcuni esempi in cui l'ID è utilizzato per implementare le funzionalità della libreria:

```
@Override
```

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mPage = getArguments().getInt(ARG_PAGE);
    setHasOptionsMenu(true);
    switch(mPage){
// ogni case implementa una parte dei listener
//(qui ne sono mostrati solo alcuni)
        case 1:
            BluetoothSPP.createBT(getActivity());

            if(!BluetoothSPP.isBluetoothAvailable()){
                FragmentActivity activity = getActivity();
                Toast.makeText(activity, "Bluetooth is not
                    available", Toast.LENGTH_LONG).show();
                activity.finish();
            }

            BluetoothSPP.setBluetoothStateListener(new
                BluetoothSPP.BluetoothStateListener() {
            public void onServiceStateChanged(int state) {
               switch (state) {
                  case Constants.STATE_CONNECTED:
                     setStatus(getString(R.string.title_connected_to,
                         BluetoothSPP.getConnectedDeviceName()));
                     break;
                  case Constants.STATE_CONNECTING:
                     setStatus(R.string.title_connecting);
                     break;
                  case Constants.STATE_NONE:
                     setStatus(R.string.title_not_connected);
                     break;
                  case Constants.STATE_DISCONNECTED:
                     setStatus(R.string.title_reconnecting);
                     break;
               }
            }
         });
            break;
        case 2:
            BluetoothSPP.setOnTemperatureDataReceivedListener(new
                BluetoothSPP.OnTemperatureDataReceivedListener() {
                @Override
                public void onDataReceived(String message) {
                    temperatureText.setText(message);
```

```
            }
        });
        break;
    case 3:
        BluetoothSPP.setOnSystolicDataReceivedListener(new
            BluetoothSPP.OnSystolicDataReceivedListener() {
            @Override
            public void onDataReceived(String message) {
                systolicText.setText(message);
            }
        });
        break;
    case 4:
        BluetoothSPP.setOnPositionDataReceivedListener(new
            BluetoothSPP.OnPositionDataReceivedListener() {
        @Override
        public void onDataReceived(String message) {
           if (Integer.parseInt(message) == 1) {
              positionImage.setImageResource(R.drawable.supine);
           } else if (Integer.parseInt(message) == 2) {
              positionImage.setImageResource(R.drawable.prone);
           } else if (Integer.parseInt(message) == 3) {
              positionImage.setImageResource(R.drawable.fawler);
           } else if (Integer.parseInt(message) == 4) {
              positionImage.setImageResource(R.drawable.
              left_lateral_recumbent);
           } else if (Integer.parseInt(message) == 5) {
              positionImage.setImageResource(R.drawable.
              right_lateral_recumbent);
           } else {
              FragmentActivity activity = getActivity();
              Toast.makeText(activity, "non-defined position",
                  Toast.LENGTH_SHORT).show();
           }
        }
     });
        break;
    case 5:
        BluetoothSPP.setOnResponseDataReceivedListener(new
            BluetoothSPP.OnResponseDataReceivedListener() {

        @Override
        public void onTemperatureDataReceived(String message) {
           requestedValue.setText(message);
```

```
            requestList.setEnabled(true);
        }

//il case 5, ovvero il quinto fragment,
//implementa tutti I listener dedicati alle richieste
        });
        break;
    }
}
```

Come si può vedere dal codice, gli switch si presta benissimo al compito di identificazione del fragment per differenziare le operazioni eseguite da essi. Per differenziare la lettura di valori singoli con la lettura di stream di dati, questi ultimo sono visualizzati utilizzando grafici mentre gli altri sono mostrati utilizzando caselle di testo. Infine, come già accennato, l'applicazione utilizza una seconda activity per la visualizzazione e selezione del bounded device:

```
package com.example.android.stabile;

import android.app.Activity;
import android.bluetooth.BluetoothDevice;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.Window;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import com.example.android.mylibrary.BluetoothSPP;
import com.example.android.mylibrary.Constants;
import java.util.Set;


public class DeviceListActivity extends Activity {


    private static final String TAG = "DeviceListActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        Log.d("DeviceListActivity", "onCreateInizio");
        super.onCreate(savedInstanceState);
```

```
        requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
        setContentView(R.layout.activity_device_list);
        setResult(Activity.RESULT_CANCELED);
        ArrayAdapter<String> pairedDevicesArrayAdapter =
                new ArrayAdapter<String>(this, R.layout.device_name);
        ListView pairedListView = (ListView)
            findViewById(R.id.paired_devices);
        pairedListView.setAdapter(pairedDevicesArrayAdapter);
        pairedListView.setOnItemClickListener(mDeviceClickListener);
        BluetoothSPP.createBT(this);
        Set<BluetoothDevice> pairedDevices =
            BluetoothSPP.getBondedDevices();
        if (pairedDevices.size() > 0) {
            findViewById(R.id.title_paired_devices).
         setVisibility(View.VISIBLE);
            for (BluetoothDevice device : pairedDevices) {
                pairedDevicesArrayAdapter.add(device.getName() + "\n"
                    + device.getAddress());
            }
        } else {
            String noDevices =
                getResources().getText(R.string.none_paired).
         toString();
            pairedDevicesArrayAdapter.add(noDevices);
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        BluetoothSPP.cancelDiscovery();
    }

    private AdapterView.OnItemClickListener mDeviceClickListener
            = new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> av, View v, int arg2,
            long arg3) {
            BluetoothSPP.cancelDiscovery();
            String info = ((TextView) v).getText().toString();
            String address = info.substring(info.length() - 17);
            Intent intent = new Intent();
            intent.putExtra(Constants.EXTRA_DEVICE_ADDRESS, address);
            setResult(Activity.RESULT_OK, intent);
            finish();
```

```
        }
    };
}
```

La selezione avvenuta su questa activity viene rilevata dall'activity principale per eseguire la connessione Bluetooth.

5.3 Implementazione di Arduino

Per modularizzare la gestione dei sensori, è stato implementato uno **scheduler** a timer che gestisce diversi task. Ogni task corrisponde a un sensore ed ha una frequenza di rilevamento differente a seconda delle caratteristiche. Di seguito il codice dello scheduler:

```
#include "Scheduler.h"
#include "Arduino.h"

void Scheduler::init(int basePeriod){
  this->basePeriod = basePeriod;
  timer.setupPeriod(basePeriod);
  nTasks = 0;
}

bool Scheduler::addTask(Task* task){
  if (nTasks < MAX_TASKS-1){
    taskList[nTasks] = task;
    nTasks++;
    return true;
  } else {
    return false;
  }
}

void Scheduler::schedule(){
  timer.waitForNextTick();
  for (int i = 0; i < nTasks; i++){
    if (taskList[i]->updateAndCheckTime(basePeriod)){
      // cli();
      taskList[i]->tick();
      // sei();
    }
  }
}
```

Ad ogni evento generato dal timer, lo scheduler verifica per ogni task che è giunto il loro momento per rilevare i valori dal sensore. Questi valori vengono letti dai sensori attraverso l'ausilio della libreria eHealth.

Ogni task implementa la classe **Task.h**, mostrata in seguito:

```
#ifndef __TASK__
#define __TASK__

class Task {
  int myPeriod;
  int timeElapsed;

public:
  virtual void init(int period){
    myPeriod = period;
    timeElapsed = 0;
  }

  virtual void tick() = 0;

  bool updateAndCheckTime(int basePeriod){
    timeElapsed += basePeriod;
    if (timeElapsed >= myPeriod){
      timeElapsed = 0;
      return true;
    } else {
      return false;
    }
  }
};

#endif
```

I metodi resi disponibili da questa classe sono:

- init: è utilizzato dal task che inizializzare le proprie variabili, collegarsi con la libreria eHealth ai sensori e impostare il periodo relativo alla frequenza di rilevamento,
- tick: racchiude le operazioni svolte dal task per la rilevazione dei dati,
- updateAndCheckTime: metodo utilizzato dallo scheduler per confrontare il periodo del task con il timer.

Di seguito è presentato un esempio di task per la gestione di un sensore (in questo caso il sensore di temperatura):

```
#include "TemperatureTask.h"
#include "Arduino.h"


TemperatureTask::TemperatureTask(Context* pContext){
  this->pContext = pContext;
}

void TemperatureTask::init(int period){
  Task::init(period);
}

void TemperatureTask::tick(){
  pContext->setTemperature(eHealth.getTemperature());
  pContext->setTemperatureState(true);
}
```

L'idea iniziale era quella di conferire ai task la possibilità di inviare direttamente i dati al client, ma le librerie scelte hanno impedito questa opzione. è stato quindi deciso di utilizzare i task per la sola lettura dei sensori, mentre l'invio dei dati è stato assegnato alla classe principale. Questa classe, ovvero **Progetto e-Health**, raccoglie le informazioni rilevate dai sensori e poi le comunica allo smartphone attraverso il modulo bluetooth. Inoltre questa classe gestisce anche il sistema di richiesta/risposta. Di seguito è mostrato il codice della classe e il collegamento del modulo bluetooth ad Arduino:

```
#include "Scheduler.h"
#include "TemperatureTask.h"
#include "AirflowTask.h"
#include "PulseOximetryTask.h"
#include "GSRTask.h"
#include "PositionTask.h"

#include <SoftwareSerial.h>
#include <eHealth.h>
#include <PinChangeInt.h>

Scheduler sched;
SoftwareSerial BTserial(A4,A5);
Context* pContext;
float temp;
```

```
char cc;
int cont = 0;

void setup(){
  Serial.begin(9600);
  BTserial.begin(9600);

  sched.init(50);
  pContext = new Context();

  Task* t0 = new TemperatureTask(pContext);
  t0->init(500);
  sched.addTask(t0);

  Task* t1 = new AirflowTask(pContext);
  t1->init(50);
  sched.addTask(t1);

  //vi e' un task per ogni sensore
}

void loop(){
  sched.schedule();
  delay(10);
  if(BTserial.available() > 0){
    cc = BTserial.read();
    Serial.println(cc);
    if(cc =='t'){
      temp = eHealth.getTemperature();
      BTserial.print("st:");
      BTserial.println(temp);
    }
    //ogni condizione if corrisponde
   //a una differente richiesta
    }
  }

  BTserial.print("a:");
  BTserial.println(pContext->getAirflow());
  if(pContext->getTemperatureState() == true){
    BTserial.print("c:");
    BTserial.println(pContext->getTemperature());
    pContext->setTemperatureState(false);
  }
```

```
  //ogni condizione if corrisponde
  //a una informazione generata dal sensore
}


void readPulsioximeter(){
  cont ++;
  if (cont == 50) {
    eHealth.readPulsioximeter();
    cont = 0;
  }
}
```

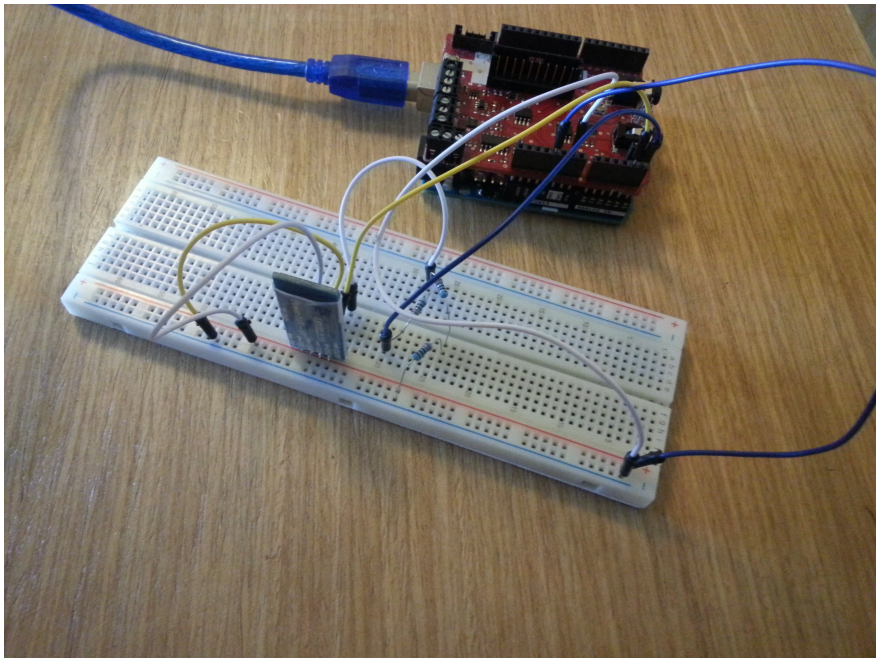


Figura 5.2: Collegamento elettrico fra Arduino e Modulo bluetooth

Per raccogliere i dati, questa classe predispone una struttura dati che viene passata come riferimento durante l'inizializzazione di ogni Task. Questa struttura è definita dalla classe **Context.h**, che rende disponibile dei metodi per l'aggiornamento dei valori. Di seguito il codice della classe:

```
#ifndef __CONTEXT__
#define __CONTEXT__
```

```
class Context {

  float temperature;
  int airflow;
  int pulse;
  int oxygen;
  float conductance;
  float resistance;
  int position;

  boolean Btemperature;
  boolean Bpulseoxygen;
  boolean Bconduresi;
  boolean Bposition;

public:

  Context(){
  }
  //metodi booleani
  void setTemperatureState(boolean state){
    Btemperature = state;
  }

  float getTemperatureState(){
    return Btemperature;
  }
  //e' presente una booleana per ogni sensore
  //metodi valori
  void setTemperature(float value){
    temperature = value;
  }

  float getTemperature(){
    return temperature;
  }
  //sono presenti metodi get/set
  //per ogni valore rilevabile dai sensori
};


#endif
```

Le variabili booleane sono utilizzare dalla classe principale per sincronizzare l'invio delle informazioni con la rilevazione delle stesse, per evitare di inviare ripetutamente informazioni ridondanti.

L'ultima annotazione da fare riguarda il task relativo ad uno specifico sensore: il pulsiossimetro. Questo è l'unico sensore che necessità dell'utilizzo di una altra libreria chiamata PinChangeInt: per far si che essa non andasse in conflitto con SoftwareSerial, si è dovuto commentare una porzione di codice di quest'ultima.

Capitolo 6

Collaudo

In questo capitolo vengono descritti tutti i test effettuati sul sistema. Innanzitutto è stata verificata la reattività della libreria a cambi di stato della connessione. Sono stati predisposti due moduli Bluetooth collegati ad Arduino e attraverso un client è stata richiesta la connessione con entrambi in successione. La libreria è stata in grado di collegarsi ad entrambi i modulo in mutua esclusione senza intoppi.

In seguito sono state verificate le API per la gestione della connessione. L'applicazione è stato sospesa e riaperta per verificare la capacità della libreria di ripristinare la connessione e il risultato è stato positivo. Inoltre è stato testato il sistema di riconnessione dovuto alla perdita di segnale: la libreria è in grado di ricollegarsi al server soltanto se lo smartphone rientra nel raggio di rilevamento entro in certo lasso di tempo.

Infine si è provato a collegare due client allo stesso server: in questo caso soltanto il primo client ad effettuare la richiesta è stato in grado di connettersi. Nel momento in cui il server collegato a un client, ogni richiesta proveniente da altri client non viene considerata.

Una volta verificate le capacità di gestione dello stato della connessione, è stata verificata la gestione della comunicazione, quindi informazioni elaborate dal server e ottenute utilizzando la libreria. Da notare che è test sono stati eseguiti utilizzando cinque dei 9 sensori a disposizione. Le API fornite hanno permesso all'applicazione di visualizzare qualunque tipo di dato, sia singolo che stream. Sporadicamente sono stati rilevati valori inconsistenti.

Infine è stato testato il sistema di richiesta/risposta. Richieste singole effettuate da un client vengono soddisfatte nella maggior parte dei casi con valori coerenti con l'informazione richiesta. Al contrario, nel momento in cui il client esegue richieste ad alta frequenza, il server non riesce a rispondere e la richiesta va perduta.

Capitolo 7

Valutazioni finali

Considerando i requisiti definiti in fase di analisi e le scelte tecnologiche effettuate, la piattaforma ottenuta rispecchia fortemente le aspettative iniziali. Tutte i requisiti funzionali specificati sono stati soddisfatti mentre alcuni di quelli non funzionali non sono stati rispettati.

La componente server si è dimostrata all'altezza delle aspettative, fornendo tutte le funzionalità di base specificate. Il servizio offerto dal server è stato più che affidabile. La fase di testing ha mostrato che la perdita di richieste è minima ed avviene solo in casi di utilizzo estremi. Inoltre le informazioni generate ed inviate sono consistenti nella maggior parte dei casi. Le soluzioni adottate in fase di progettazione per affrontare la scarsa capacità computazionale sono state quelle giuste. Per poter applicare anche le funzionalità descritte nei requisiti non funzionali sarà necessario fare qualche accorgimento nelle scelte tecnologiche.

Il protocollo di comunicazione costruito per il sistema ha svolto molto bene il suo compito, mentre la libreria risulta essere molto reattiva ai cambi di situazione. La fase di testing ha mostrato che la libreria è in grado di gestire ogni cambio di stato della connessione in maniera soddisfacente. Anche in questo caso, un cambio di tecnologia permetterebbe di aumentare le capacità della libreria aggiungendo qualche linea di codice.

L'applicazione prototipo sviluppata per il sistema ha permesso di visualizzare tutte le informazioni in maniera corretta. Le API fornite dalla libreria hanno permesso di ottenere in feedback continuo dello stato della connessione e di usufrui di tutte le funzionalità di base specifica.

Concludendo, il sistema ottenuto fornisce un ottimo ambiente di sviluppo per applicazioni mobile inerenti alla rilevazione di parametri vitali. Inoltre fornisce una base molto solida per eventuali sviluppi futuri, con la possibilità di cambiare l'interfacciamento coi sensori e utilizzando dispositivi come smartglass o simili.

Conclusioni

Lo sviluppo di questo sistema è stata una esperienza unica dal mio punto di vista. Fin da subito ho avuto a che fare con tematiche nuove che mi hanno motivato ad imparare. Progettare quasi da zero un sistema basato su una idea non è stato facile ed ha richiesto una grande quantità di documentazione, che mi ha permesso di imparare aspetti nuovi dell'informatica e di approfondire l'utilizzo di molto tecnologie. Concetti come Wearable computing, programmazione mobile e healthcare ora fanno parte del mio bagaglio culturale.

Il sistema ottenuto durante questo percorso ha rispecchiato quasi tutti i requisti e gli obiettivi prefissati. Inoltre la scalabilità del sistema permetterà modifiche futuri che saranno descritte nel prossimo paragrafo.

Sviluppi Futuri

Nell'immediato futuro sarà possibile adattare la libreria a più sistemi operativi ed implementare tutte quelle funzionalità proposte nei requisiti non funzionali che non è stato possibile adottare per mancanza di tempo e per le scelte tecnologiche effettuate. Sarebbe interessante cambiare il sistema di comunicazione ed adottare una connessione wi-fi: adattando la libreria si potrebbe introdurre la possibilità di connettere più client al server contemporaneamente.

Per quanto riguarda il server, è possibile cambiare l'unità computazione con una più avanzata, come RaspberryPI. In questo modo è possibile migliorare ulteriormente la reattività del sistema ed aggiungere ulteriori funzionalità. Sarebbe interessante poter configurare il sistema in base alle proprie preferenze. Inoltre, si potrebbero aggiungere i sensori che non sono stati utilizzando, controllando eventuali conflitti interni causati dalle librerie utilizzate.

Una aggiunta importante che aumenterebbe le capacità del server riguarda un sistema di archiviazione dei dati. Si potrebbe predisporre un database per il salvataggio dei dati rilevati ad un paziente per avere un resoconto dello stato di salute della persona per corso di un certo periodo. Adottando la tecnologia wi-fi non è necessario aggiungere altri sistemi di comunicazione.

Infine, si potrebbe creare un sistema di interfaccia avanzato utilizzando degli smarglass: l'idea è quella di collegare il dispositivo wearable sul paziente e collegarsi a esso utilizzando gli smartglass. In questo modo è possibile vedere i parametri vitali del paziente guardando attraverso i glass.

Concludente, data la scalabilità del sistema, è possibile aggiungere moltissime funzionalità e piccoli dettagli per migliorare le potenzialità che questo dispositivo offre all'utilizzatore.

Ringraziamenti

Ringrazio la mia famiglia che mi ha permesso di frequentare questa università.

Ringrazio il prof. Alessandro Ricci per avermi dato l'opportunità di sviluppare questo progetto e il co-relatore Angelo Croatti per i grandi consigli che mi ha dato durante lo sviluppo del progetto.

Infine ci tengo a ringraziare i miei amici che mi hanno sostenuto durante tutto il percorso di stesura di questa tesi.

Bibliografia

[1] Thad Starner, *The Challenges of Wearable Computing: Part1*, 2001.

[2] Thad Starner, *The Challenges of Wearable Computing: Part2*, 2001.

[3] Kalpesh A. Popat, Dr. Priyanka Sharma, *Wearable Computer Applications A Future Perspective*, 2013.

[4] Samuel Mann, *Wearable computing as a means for personal empowerment*, 1994.

[5] Joseph Wei, *How Wearables Intersect with the Cloud and the Internet of Things*, 2014.

[6] Yao meng, Hee-Cheol Kim, *Wearable Systems and Applications for Healthcare*, 2011.

[7] Yao Meng, Heung-Kook Choi, *Exploring the User Requirements for Wearable Healthcare Systems*, 2011.

[8] P. Lukowicz1, T. Kirstein, G. Tröster, *Wearable systems for health care applications*, 2004.

[9] Melanie Swan, *Sensor Mania! The Internet of Things, Wearable Computing, Objective Metrics, and the Quantified Self 2.0*, 2012.

[10] Wikipedia, *Wearable computer*, 2015.

[11] Katherine Watier Ong, *Marketing Wearable Computers to Consumers*, 2003.

[12] Steve Mann, *Wearable computing*, 2012.

Elenco delle figure