ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

IMPLEMENTAZIONE DI UN SISTEMA PER IL SELF-MANAGEMENT DEL DIABETE DI TIPO 1

Tesi in:
Sistemi Autonomi

Relatore:
Chiar.mo Prof.
ANDREA OMICINI
Co-relatore:
Dott.ssa SARA MONTAGNA

Presentata da:
FRANCESCO COZZOLINO

I SESSIONE
ANNO ACCADEMICO 2017–2018

PAROLE CHIAVE

Mobile Health

Internet of Things

Self-Management di Malattie Croniche

MASON

Diabete

Introduzione

Con il forte progresso tecnologico degli ultimi anni si è assistito a una rapida diffusione della visione dell'*Internet of Things*. Tale visione prevede la presenza di oggetti del mondo reale collegati in rete, in grado di interagire tra di essi o con l'essere umano, al fine di migliorare la vita di quest'ultimo.
Gli ambiti in cui può essere applicata la visione dell'Internet of Things sono molteplici e, tra questi, è da segnalare l'ambito dell'*Healthcare*.
L'Healthcare, grazie al progresso tecnologico, ha subito un processo di digitalizzazione e, oggi, grazie alla rapida diffusione dell'Internet of Things e alla presenza degli smartphone, si sta diffondendo il concetto di *mobile Health*. Con questo termine si intende la realizzazione di servizi in ambito sanitario, attraverso l'utilizzo di device mobili, in grado di aggirare i limiti temporali e geografici.
Il mobile Health consente di migliorare o cambiare il modo in cui vengono forniti i servizi sanitari, offrendo l'opportunità all'assistito di monitorare e gestire in modo autonomo il proprio stato di salute. La pratica con la quale una persona svolge delle operazioni in totale autonomia, atte a migliorare il proprio stato di salute, prende il nome di *self-Management*.

Il diabete mellito è una malattia cronica che causa un'elevata concentrazione di glucosio nel sangue, superiore ai valori di riferimento, per lunghi periodi di tempo. La convivenza con questa patologia cronica può causare delle problematiche nella vita quotidiana. Al fine di ridurre le problematiche, oltre a seguire le prescrizioni mediche, degli accorgimenti sull'alimentazione e sull'attività fisica possono risultare fondamentali.
Le persone affette da diabete, che decidono di intraprendere un percorso in cui adottano uno stile di vita più salutare, necessitano di continue informazioni sul proprio stato di salute; al fine di prendere la migliore decisione possibile.
L'obiettivo di questa tesi, pertanto, è quello di realizzare un sistema in grado

di fornire gli strumenti necessari per il self-Management del diabete mellito di tipo 1, adottando la visione dell'Internet of Things e del mobile Health.
Nel dettaglio, si vuole realizzare un sistema che consenta all'assistito di simulare il proprio andamento glicemico attraverso il proprio smartphone; con lo scopo di ricevere dei consigli sull'alimentazione e sull'attività fisica da seguire.

Il primo capitolo tratterà la tematica dell'Internet of Things, illustrandone i contesti di maggiore interesse in cui può trovare applicazione, l'architettura tipica di un sistema che adotta tale visione, le tecnologie che ne consentono la realizzazione e le principali sfide.

Il secondo capitolo si focalizzerà sul mobile Health, illustrandone l'architettura, i benefici che ne derivano dall'applicazione di questa visione e le principali sfide attualmente presenti.
All'interno di questo capitolo verrà inoltre fornita una panoramica sul self-Management.
Il capitolo si concluderà con una presentazione di alcune applicazioni nell'ambito del mobile Health.

Il terzo capitolo si occuperà di presentare il progetto, l'architettura logica e le tecnologie selezionate per la realizzazione del sistema. Verrà inoltre fornita una panoramica sul metabolismo del glucosio; al fine di fornire le conoscenze basilari relative al diabete mellito di tipo 1.

Il quarto, il quinto e il sesto capitolo presenteranno i dettagli implementativi di maggiore rilevanza delle singoli componenti del sistema.

Infine, Il capitolo conclusivo di questa trattazione, presenterà i possibili sviluppi futuri che potranno servire a completare il sistema.

Indice

Capitolo 1

Internet of Things

Dagli anni 80' fino ad oggi, lo sviluppo tecnologico ha reso possibile la realizzazione di device con capacità computazionali sempre maggiori e di dimensioni sempre più ridotte. Inoltre, grazie a Internet, a nuove tecnologie di comunicazione e a batterie dotate di una buona autonomia in dimensioni contenute, tali device sono diventati parte integrante della nostra vita; consentendo la diffusione di una nuova visione: l'*Internet of Things* (IoT). Secondo tale paradigma, gli oggetti presenti nella vita quotidiana, sono collegati in rete e sono identificabili univocamente. Tali oggetti sono in grado di percepire l'ambiente in cui sono situati e interagire tra di essi attraverso l'ausilio di reti.
Si stima che per il 2020 si possa arrivare ad avere circa 212 miliardi di entità connesse in rete e, l'interazione M2M (Machine To Machine), possa raggiungere entro il 2022 il 45% del traffico Internet [1].

Il seguente capitolo si apre con una presentazione degli ambiti di applicazione di maggiore interesse della visione IoT.
Il secondo paragrafo illustra l'architettura tipica di un sistema IoT, seguito da una sezione con le tecnologie che permettono la realizzazione di sistemi che adottano tale visione.
Il terzo paragrafo offre una panoramica sulle sfide attualmente presenti nell'applicazione della visione IoT.
Il capitolo si conclude con un approfondimento dell'IoT nell'ambito dell'Healthcare; in quanto oggetto di interesse di questa tesi.

1.1 Applicazioni

Oggigiorno l'IoT è presente in modo consolidato in alcuni ambiti, ed è destinato a crescere nel prossimo futuro in diversi settori. L'obiettivo dei sistemi che adottano questa visione, indipendentemente dal settore in cui trovano applicazione, è uno solo: migliorare la qualità della vita dell'uomo.
Di seguito sono presenti gli ambiti in cui l'IoT è ormai una realtà consolidata o che è destinato a rivestire un ruolo importante nel prossimo futuro.

1.1.1 Healthcare

Si stima che nel 2025, tra i vari ambiti in cui l'IoT troverà applicazione, l'healthcare avrà un impatto economico del 41% [1].
Grazie alla possibilità di avere dei sensori, soprattutto wearable, è possibile monitorare in qualsiasi momento e, in ogni luogo, i segnali vitali di una persona.
Uno dei contesti in cui possono essere utilizzati maggiormente è il monitoraggio dei pazienti affetti da malattie croniche.
L'IoT oggigiorno è presente soprattutto in ambito wellness, consentendo alle persone di monitorare, attraverso wristband, alcuni segnali vitali; al fine di migliorare in modo autonomo il proprio stato di salute.

1.1.2 Logistica

La logistica è uno dei primi settori in cui l'oT ha trovato applicazione.
Le aziende del settore, al fine di ottimizzare i processi di spedizione, hanno la necessità di monitorare ogni singola spedizione; offrendo, inoltre, al cliente l'opportunità di monitorare lo stato dell'ordine e il luogo in cui si trova.
Una delle prime tecnologie che ha permesso di introdurre l'IoT in questo contesto è l'RFID (Radio Frequency IDentification). Tale tecnologia si basa su dei tag al cui interno è presente un microchip in grado di scambiare delle informazioni.
L'RFID consente di realizzare dei tag di dimensioni ridotte e con un costo di produzione basso. Pertanto, è possibile applicare dei tag sui colli e, una volta giunti negli hub delle aziende di logistica, attraverso un lettore, è possibile raccogliere le informazioni inerenti al collo. In questo modo, il cliente è in grado monitorare lo stato del proprio ordine, mentre l'azienda di logistica, sulla base

dei dati raccolti, è in grado di prendere le migliori decisioni al fine di ridurre i costi e i tempi di trasporto.

1.1.3 Smart city

Con il termine smart city si denota un sistema formato da infrastrutture di comunicazione avanzate e servizi realizzati nell'ambito delle città.
Nelle smart cities vi sono diverse aree di applicazione della visione IoT. Di seguito sono presenti quelli di maggiore interesse.

Smart Home

Nell'ambito delle smart city è attualmente il settore più sviluppato. Oggigiorno, in ogni casa è presente una smart Tv e, sul mercato, sono presenti ormai piccoli e grandi elettrodomestici in grado di comunicare con gli smartphone. Altri esempi concreti di oggetti smart presenti nelle nostre case possono essere: robot aspirapolvere, lampadine, valvole elettrostatiche e sistemi di videosorveglianza. Inoltre, al fine di rendere più completo l'ecosistema che è possibile realizzare in casa, è necessario segnalare la comparsa sul mercato degli assistenti vocali; attraverso i quali è possibile controllare gli oggetti smart presenti all'interno della casa.

Automotive

L'automotive rappresenta uno dei settori agli albori e di maggiore interesse; data la presenza di alcune soluzioni e gli investimenti ingenti da parte delle aziende del campo.
I veicoli attualmente in commercio permettono alle persone, attraverso il proprio smartphone, di visualizzare alcune informazioni, quali: stato della batteria (per i veicoli elettrici), posizione del veicolo attraverso il gps, informazioni sul viaggio effettuato e stato delle componenti meccaniche (usura gomme, freni, etc).
Al momento sono in fase di ricerca e sviluppo i cosiddetti protocolli *Vehicle To Vehicle* (V2V). Tali protocolli hanno l'obiettivo di permettere ai veicoli di comunicare fra di essi. In un sistema a guida autonoma, ad esempio, per i veicoli di flotte commerciali, è possibile farli interagire al fine di regolare la velocità per ridurre i consumi. Inoltre, tale protocollo può essere utilizzato per

permettere lo scambio di informazioni tra i veicoli, come ad esempio: lo stato di usura delle gomme, dei freni e di altre componenti; in modo da permettere al veicolo che segue di regolare la distanza di sicurezza da mantenere.
Con l'aumentare del numero di veicoli connessi a Internet, inoltre, vi è la possibilità di acquisire una maggiore quantità di dati sul traffico; al fine di permettere ai mezzi di emergenza di raggiungere il sito di interesse attraverso un percorso ottimale.

Smart Grid

Per smart grid si intende solitamente una rete elettrica costituita da sensori e attuatori adibiti alla raccolta e analisi dei dati. In questo ambito, con l'utilizzo di sensori che inviano informazioni sui consumi elettrici nei quartieri, è possibile ottimizzare l'utilizzo e la distribuzione dell'energia elettrica.

1.2 Architettura

L'architettura di un tipico sistema IoT è costituito da almeno tre livelli: *perception*, *network* e *application*.
Il livello di *perception* è costituito dai trasduttori che si trovano nel mondo reale. I sensori, in questo livello si limitano a raccogliere informazioni sull'ambiente o sul particolare oggetto (temperatura, umidità, accelerazione, etc). Le informazioni raccolte a questo livello vengono trasmesse al livello di network. Gli attuatori, invece, svolgono delle azioni che possono modificare l'ambiente in cui si trovano, attraverso le informazioni ricevute dal livello di network.
Il livello di *network* si occupa di trasferire le informazioni tra il livello di applicazione e il livello di perception.
Il livello di *applicazione* si occupa di gestire e processare i dati al fine di garantire il servizio realizzato.
Tra il livello di network e di applicazione, può essere presente un livello logico intermedio, detto di *middleware*. In questo livello si collocano tutte le tecnologie e/o tecniche che permettono di gestire la rete e i dati che vi circolano. Questo livello può essere stratificato su diversi network, e può essere presente nei dispositivi di livello inferiore dotati di sufficiente capacità computazionale. Tale livello può essere particolarmente utile quando il sistema fornisce più servizi e i trasduttori che realizzano il servizio devono comunicare solo tra di essi. Sopra al livello di applicazione, in alcuni sistemi, può essere aggiunto un ulte-

riore livello logico, detto di *business*; il quale si occupa di elaborare e analizzare i dati prodotti dal sistema.

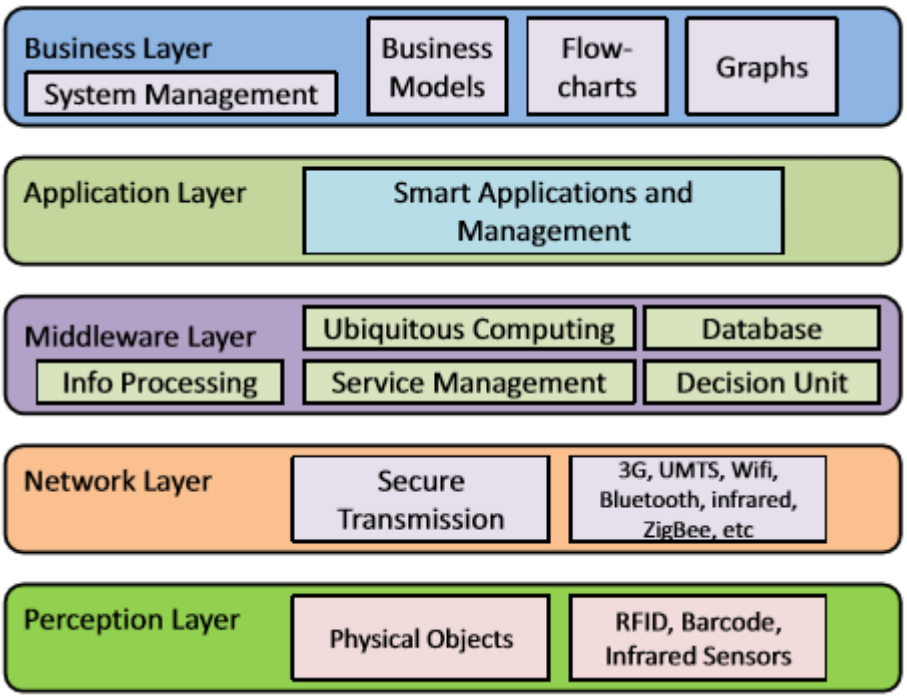

Figura 1.1: Architettura di un sistema IoT[2]

1.3 Tecnologie abilitanti

Oggigiorno sono diversi i protocolli che consentono la realizzazione di sistemi che adottano la visione IoT. Prendendo come riferimento lo stack TCP/IP, i protocolli più diffusi in ambito IoT si possono collocare nei seguenti livelli: *fisico*, *internet* e *applicazione*.

1.3.1 Livello fisico

Bluetooth e Bluetooth Low Energy

Bluetooth è uno standard per la trasmissione di dati utilizzato all'interno di WPAN (Wireless Personal Area Network). Si tratta di uno standard che consente lo scambio di dati attraverso le frequenze radio a corto raggio. Questa tecnologia di comunicazione è presente da circa 20 anni e, nel corso del tempo, sono state rilasciate diverse versioni. Dal 2010, soprattutto con l'immissione sul mercato di device il cui problema era garantire un buon livello di autonomia, è stato rilasciato Bluetooth 4.0, denominato anche Bluetooth Low Energy (BLE). Tale versione consente di ridurre drasticamente i consumi energetici da parte dei device che utilizzano questa tecnologia di comunicazione. Pertanto, risulta essere ideale per la realizzazione di sistemi IoT, data la necessità da parte dei sensori e degli attuatori di ridurre al minimo i consumi.
Il raggio di copertura per la trasmissione dei dati è di circa dieci metri.

ZigBee

Si tratta di un protocollo di comunicazione basato sullo standard IEEE 802.15.4. La trasmissione dei dati avviene attraverso onde radio a corto raggio, con una frequenza di 2.4 GHz. A differenza di Bluetooth e BLE, il raggio di copertura si estende fino a 100 metri, ma la velocità di trasmissione risulta essere leggermente inferiore.

WiFi

Il WiFi è una rete wireless che si basa sullo standard IEEE 802.11, utilizzato per le reti locali senza fili. Lo standard 802.11 definisce diverse classi e, per ognuna di essa, variano la velocità e la frequenza di trasmissione. Di seguito l'elenco delle classi attualmente presenti, con le relative velocità di trasmissione e frequenze a cui operano:

- **Classe a:** 54 Mb/s - 5 GHz
- **Classe b:** 11 Mb/s - 2.4 GHz
- **Classe g:** 54 Mb/s - 2.4 GHz

- **Classe n:** 450 Mb/s - 2.4/5 GHz
- **Classe ac:** 3 Gb/s - 5 GHz

All'aumentare della velocità di trasmissione, aumentano anche i consumi dei dispositivi. Pertanto, gli standard consigliati al fine di ridurre i consumi energetici sono l'802.11b e l'802.11g.

Ethernet

L'Ethernet si basa sullo standard IEEE 802.3 e consente di realizzare LAN cablate. Trova applicazione nei contesti in cui i device non sono in grado di trasmettere i dati utilizzando le tecnologie wireless, oppure, nei contesti in cui l'ubicazione dei device non rendono possibile l'utilizzo di tali tecnologie.

LTE

Le reti cellulari come l'LTE (Long Term Evolution), possono essere utilizzate nel contesto dell'IoT. Come per il WiFi, anche per l'LTE sono presenti diverse classificazioni; in cui ogni classe ha caratteristiche diverse.
L'LTE-M risulta essere adatta nei contesti in cui bisogna minimizzare i consumi energetici, data la velocità di trasmissione limitata a 1 Mbit/s.
La rete 5G, destinata a comparire nell'immediato futuro, viene indicata come la futura tecnologia di comunicazione per i sistemi IoT.

LoRa

Si tratta di una tecnologia di comunicazione wireless a basso consumo per reti LPWAN (Low Power Wide Area Network). La velocità di trasmissione dati è piuttosto bassa, compresa tra i 0,3 kbps e i 50 kbps, ma garantisce un'ampia copertura, compresa tra i 15 e i 20 km. Data l'ampia copertura, tale protocollo consente di coprire delle vaste aree con un'infrastruttura abbastanza ridotta.
Si tratta di una tecnologia compatibile con lo standard IPv6.

NFC

L'NFC (Near Field Communication) è una tecnologia di comunicazione a corto raggio basata su onde radio, con un raggio di comunicazione di qualche centimetro.

Tipicamente utilizzato per i pagamenti, è molto presente sugli smartphone. Permette di scambiare piccoli quantitativi di dati con un basso consumo energetico.

RFID

L'RFID (Radio Frequency IDentification) è una tecnologia che consente la memorizzazione di dati e l'identificazione di oggetti attraverso dei tag al cui interno è presente un chip. Lo scambio di dati si basa su onde radio a corto raggio.
Il raggio di comunicazione di questa tecnologia è limitato a qualche centimetro. I tag RFID non consentono solo di leggere informazioni, ma anche di scriverle. Esistono tre tipi di tag:

- **Attivi:** sono alimentati da batterie e lo scambio di informazioni avviene quando in prossimità del tag è presente un lettore RFID
- **Semi-Attivi:** la batteria è utilizzata solo per mantenere attiva la parte circuitale, mentre per la trasmissione dei dati viene impiegata parte dell'energia ricevuta dall'onda radio del lettore RFID
- **Passivi:** l'energia viene ricavata esclusivamente dalle onde radio emesse dal lettore RFID

1.3.2 Livello Internet

IPv6

In questo livello i device sono identificati da indirizzi IP. IPv6 è il protocollo internet designato come successore dell'IPv4. Nell'IoT tale protocollo è fondamentale, in quanto lo spazio degli indirizzi assegnabili da IPv4, essendo a 32 bit, è di 4.3 miliardi. I device IoT attualmente presenti, superano questo numero; pertanto è necessario uno spazio degli indirizzi maggiore. IPv6 utilizza uno spazio degli indirizzi a 128 bit. Pertanto, il possibile numero di indirizzi assegnabili è di 2^{38}.

6LoWPAN

Protocollo che permette l'adozione di IPv6 sulle reti wireless che adottano lo standard IEEE 802.15.4. Si tratta di un protocollo in cui vengono adottati

meccanismi di compressione dei pacchetti; al fine di poterli trasmettere all'interno di reti basate sullo standard IEEE 802.15.4.
Poiché la dimensione dei pacchetti inviati è ridotta, il consumo energetico necessario per l'invio risulta essere basso; rendendo questo protocollo adatto per la realizzazione di sistemi IoT.

RPL

Si tratta di un protocollo di routing del traffico per reti a bassa potenza; realizzato per instradare il traffico delle reti basate sullo standard IEEE 802.15.4. È progettato per instradare pacchetti in reti in cui i dispositivi non sono sempre disponibili e dove ci può essere un'elevata perdita dei pacchetti.
Al fine di ridurre al minimo il consumo energetico dei device, RPL per determinare il routing dei pacchetti, costruisce un grafo dei device presenti nella rete.

1.3.3 Livello applicazione

HTTP/HTTPS

I protocolli HTTP/HTTPS possono essere utilizzati per realizzare servizi di tipo REST, sebbene non siano ritenuti i più adatti nell'ambito IoT per due motivi:

- In relazione al contenuto del body di una richiesta HTTP, vengono inviate informazioni aggiuntive non necessarie per il sistema, presenti tipicamente nell'header
- La quantità di dati da manipolare per i device smart a volte può essere problematica

MQTT

MQTT (Message Queue Telemetry Transport) è un protocollo concepito per essere utilizzato nei contesti in cui la banda è poca e la latenza della rete è elevata. La comunicazione tra i device avviene attraverso dei messaggi, adottando il pattern publish-subscribe.
Data la peculiarità per cui è stato realizzato questo protocollo, risulta essere adatto per l'utilizzo in reti non affidabili e/o da parte di trasduttori.

AMQP

AMQP (Advanced Message Queuing Protocol) è un protocollo open per i message oriented middleware. Tale protocollo è concepito per consentire l'invio di messaggi attraverso un middleware; un'infrastruttura che permette lo scambio di messaggi tra due entità attraverso un layer uniforme, garantendo dunque l'interoperabilità.
Il middleware realizzato sulla base delle specifiche di questo protocollo garantisce: la gestione dei messaggi, la funzionalità di reporting degli errori, la persistenza dei messaggi, il disaccoppiamento dei device, l'affidabilità e la scalabilità del sistema.
Poiché la comunicazione tra i partecipanti si basa sullo scambio di messaggi, il middleware, per garantire l'ordine di quest'ultimi, impiega delle code.
I metodi offerti dal middleware per lo scambio dei messaggi sono:

- **Point to Point:** invio asincrono dei messaggi tra due entità
- **Publish - Subscribe:** comunicazione one to many o many to many. Il publisher invia i messaggi a un'entità detta exchange, la quale si occupa del dispatch del messaggio, inviandolo alle code delle entità interessate a ricevere un particolare tipo di messaggio e/o da una particolare entità

XMPP

XMPP (Extensible Messaging and Presence Protocol) è un protocollo message oriented middleware, basato sullo scambio di piccole porzioni di dati strutturati sul formato XML. Si basa su un'architettura client-server decentralizzata, in cui i client non comunicano direttamente tra di loro, ma i messaggi vengono veicolati dai server. Trattandosi di un modello decentralizzato, chiunque può essere un server.

CoAP

CoAP (Contrained Application Protocol) è nato per consentire l'adozione di HTTP su device con risorse limitate o in reti con scarsa banda o con un'elevata perdita di pacchetti. Si può ritenere una versione leggera di HTTP; in quanto l'header ha una dimensione che può variare tra i 10 byte e i 20 byte. Inoltre, il protocollo utilizzato a livello di trasporto è UDP.

A differenza di AMQP, MQTT e XMPP, tale protocollo prevede un'architettura client-server centralizzata.

1.4 Sfide dell'IoT

Dal momento in cui è stato introdotto il concetto di IoT, fino alla realizzazione dei primi sistemi, sono intercorsi alcuni anni; in quanto vi erano dei problemi legati all'hardware e alle tecnologie di comunicazione non ancora mature. Questi problemi si possono ormai considerare risolti, ma ancora oggi sono presenti molte sfide. Di seguito è possibile trovare le sfide secondo [1] e alcune possibili soluzioni.

Disponibilità

Il problema della disponibilità rappresenta una sfida per qualsiasi tipo di sistema, non solo per i sistemi IoT.
I sistemi realizzati devono essere sempre disponibili. Nel caso dell'IoT, trattandosi di oggetti presenti nel mondo reale, è necessario garantire la disponibilità sia a livello software che a livello hardware.
Una soluzione a questo problema potrebbe essere quella di garantire la ridondanza per i device e i servizi considerati critici.

Affidabilità

Per affidabilità si intende la capacità di un sistema di funzionare sulla base delle specifiche per cui è stato realizzato. L'affidabilità ha lo scopo di incrementare la percentuale di successo nella fornitura dei servizi. Affidabilità e disponibilità sono spesso correlati, in quanto un servizio che tende a essere non disponibile, al tempo stesso è poco affidabile.
Per creare un servizio IoT affidabile, bisogna considerare diversi aspetti a seconda del dominio applicativo del sistema. Può essere fondamentale l'accuratezza dei dati acquisiti dai device, la latenza di trasmissione, il tempo di esecuzione di una routine o la perdita di pacchetti nella trasmissione dei dati. In un sistema con vincolo hard real time, ad esempio, bassa latenza e tempo di esecuzione sono fondamentali.
Nella realizzazione di un sistema IoT, nel caso in cui possa interagire con delle persone, bisogna garantire che quest'ultimo non possa nuocere all'essere

umano in alcun modo. Si pensi ad esempio a un sistema di healthcare; qualora quest'ultimo non dovesse essere affidabile, potrebbe portare alla morte del paziente.

Performance

A causa delle dimensioni ridotte, la capacità computazionale dei device smart è limitata. Si pone dunque la sfida di realizzare sistemi efficienti che sfruttano al meglio l'hardware sottostante e, che al tempo stesso, siano in grado di fornire un servizio affidabile ed efficiente.

Scalabilità

Per scalabilità si intende la capacità di ampliare o ridurre il numero di device, servizi e funzionalità in base al fabbisogno; senza influenzare la qualità del servizio attualmente offerto.
Nell'IoT, a causa dell'eterogeneità dei device e dei numerosi protocolli di comunicazioni esistenti in questo ambito, è difficile realizzare delle architetture scalabili.

Interoperabilità

L'interoperabilità tra i device è forse la sfida più difficile nella realizzazione di questo tipo di sistemi, soprattutto per uso privato. Attualmente vi è una pletora di protocolli di comunicazione e, dunque, si pone la sfida di permettere ai device di comunicare tra di loro e di integrare i sistemi.
Al momento, più che una sfida, si tratta di una barriera realizzata da chi fornisce soluzioni IoT, in quanto, per consentire l'utilizzo dei propri ecosistemi, i produttori tentano di impedire l'integrazione con dispositivi non realizzati da essi. Si pensi ad esempio a una cosa domotica. Difficilmente sarà possibile utilizzare un termostato smart di un costruttore, abbinato con delle elettrovalvole realizzate da un fornitore differente.
Una possibile soluzione potrebbe essere quella di cambiare la visione del paradigma e passare dall'Internet of Things, al Web of Things; dove i device comunicano attraverso dei servizi REST.

Sicurezza e Privacy

Garantire la sicurezza di un sistema e la privacy degli utenti è una delle sfide più ardue nella realizzazione dei sistemi. Nel contesto dell'IoT, tale sfida è ancora più ardua, in quanto possono essere presenti anche milioni di device che interagiscono tra di loro o, addirittura, device che interagiscono con l'ambiente, attuando una serie di azioni in grado di modificarlo. In quest'ultimo caso, la sicurezza dei device è di vitale importanza.

1.5 Internet of Things nell'Healthcare

Uno degli ambiti di maggiore interesse in cui può essere applicato il paradigma dell'IoT è l'Healthcare; sia per uso privato che enterprise. La miniaturizzazione delle componenti elettroniche, abbinate con questa visione, consentono la realizzazione di nuove tipologie di servizi o di migliorarne alcuni.
L'IoT può trovare applicazione per fornire una maggiore e migliore assistenza diretta ai pazienti, utilizzando sistemi che prevedono il coinvolgimento diretto dell'assistito, oppure può essere fornita un'assistenza indiretta, adottando soluzioni che non coinvolgono in prima persona il paziente.

1.5.1 Assistenza indiretta ai pazienti

Queste soluzioni trovano applicazione prevalentemente all'interno delle strutture sanitarie, al fine di fornire la migliore degenza possibile ai pazienti ricoverati. In una struttura ospedaliera, i rischi di infezione sono elevati e possono causare complicazioni o addirittura la morte dei degenti. Al fine di ridurre i rischi di infezione, si possono quindi adottare soluzioni che prevedono l'utilizzo di sensori in grado di misurare il livello di sterilità dell'ambiente; al fine di intraprendere, se necessario, le giuste misure per garantire un adeguato livello di sterilità dei locali.
Il personale sanitario, potenzialmente, rappresentano dei vettore di infezioni. Pertanto, l'igiene di quest'ultimi è di vitale importanza. Una soluzione potrebbe prevedere l'utilizzo di un braccialetto dotato di tecnologia RFID e la presenza di dispenser igienizzanti dotati di relativo lettore. In questo modo, si potrebbe tener traccia del personale sanitario e il tempo trascorso dall'ultima volta in cui hanno sterilizzato le proprie mani.
L'IoT può essere inoltre utilizzato per effettuare manutenzione predittiva dei

macchinari presenti in una struttura ospedaliera, intervenendo per tempo con la riparazione e/o la sostituzione; evitando situazioni di malfunzionamento e rallentamento dei servizi erogati.
L'IoT, infine, può trovare impiego anche per lo stoccaggio nelle celle frigorifere dei medicinali e dei vaccini, adottando soluzioni che siano in grado di mantenere una temperatura costante, fornendo il numero di scorte disponibili per ogni tipologia di farmaco e notificandone inoltre l'esaurimento; al fine di evitare delle emergenze dovute alle carenze di quest'ultimi [3].

1.5.2 Assistenza diretta ai pazienti

Uno degli ambiti in cui può trovare sicuramente applicazione la visione dell'IoT è senza dubbio il monitoraggio remoto.
Nei sistemi di monitoraggio tradizionali, al paziente vengono applicati dei sensori sul corpo; i quali memorizzano i dati raccolti su un device di storage. Il paziente deve convivere con dei sensori cablati per alcuni giorni e, una volta terminata la raccolta dei dati, vengono analizzati dal medico al fine di produrre una diagnosi. Questo tipo di monitoraggio presenta una serie di svantaggi per tutti gli attori coinvolti. Il paziente è costretto a convivere con dei sensori cablati e la sua mobilità risulta ridotta. Il medico, invece, è costretto ad attendere dei giorni prima di visualizzare i dati e analizzarli, ritardando dunque l'esito della diagnosi. Inoltre, con questa tipologia di sistemi, vengono forniti al medico, in modo grezzo, tutti i dati prodotti durante l'attività di monitoraggio. La notevole quantità di dati prodotti, può dunque risultare dispersiva e indurre il medico all'errore.
Grazie all'IoT, invece, il monitoraggio diventa remoto e, grazie alle tecnologie wireless, il paziente non è costretto a indossare dei sensori cablati, permettendo dunque all'assistito di compiere le azioni quotidiane senza l'intralcio di cavi. Inoltre, i sensori sono in grado di comunicare direttamente con la struttura sanitaria, permettendo dunque l'analisi dei dati in tempo reale da parte del medico.
Con la recente diffusione dell'intelligenza artificiale, inoltre, i dati raccolti dai sensori possono essere processati al fine di mostrare le informazioni più significative e, nel caso in cui il medico dovesse richiederlo, potrà visualizzare i dati con un livello di dettaglio maggiore; riducendo di fatto il margine di errore.

Body Area Network

Nei sistemi realizzati per fornire assistenza diretta ai pazienti, l'IoT trova applicazione soprattutto nelle *Body Area Network* (BAN). Con questo termine si intende una rete che interconnette dei sensori e/o attuatori smart dotati di un limitato raggio di copertura.
Molto spesso tali device non sono dotati di un accesso diretto a Internet per diversi motivi. Spesso i trasduttori sono di dimensioni ridotte; in quanto devono essere posizionati sul corpo o, in alcuni casi, impiantati all'interno. Le dimensioni ridotte limitano le capacità funzionali del device e l'autonomia della batteria. Inoltre, poiché i trasduttori si trovano a contatto con il corpo, le emissioni radio risultano essere dannose per la salute dell'uomo. Infine, nei casi di sistemi con vincoli hard-real time, è necessario minimizzare il ritardo della comunicazione, pertanto Internet non risulta essere adatto.
Per fornire un accesso Internet ai device di una BAN, si utilizza un *Body Gateway*. Il gateway si occupa di coordinare i device della BAN e, solitamente, interagisce con un *Network Hub* (ad esempio via Bluetooth); il quale fornisce l'accesso a Internet.
Nonostante la possibilità di una BAN di interagire via Internet, essa deve essere in grado di operare anche in contesti privi di accesso a Internet.

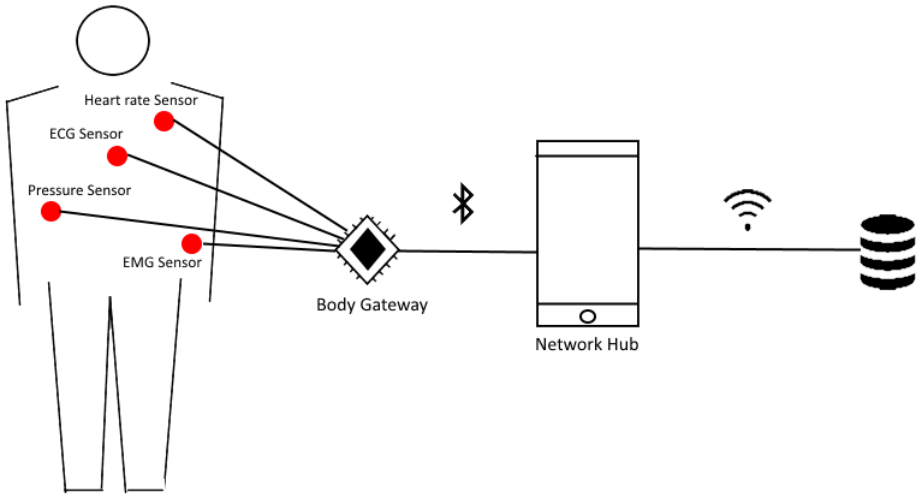

Figura 1.2: Architettura di una BAN

In letteratura le BAN sono un tema molto affrontato, ma è difficile trovare delle applicazioni reali a causa delle sfide non ancora risolte. Per le BAN in cui i sensori sono dotati di cavi, quest'ultimi possono essere di intralcio per la persona e ostacolarla nelle azioni quotidiane. Per i sensori dotati di tecnologia wireless, invece, si pone il problema delle interferenze tra i sensori e/o altre BAN. Infine, sono presenti le problematiche tipiche dell'IoT, quali: la sicurezza dei dati, la loro gestione e l'interoperabilità tra i device.

Capitolo 2

Mobile Health e Self-Management

Il seguente capitolo offre una panoramica sul *mobile Health* e sul *self-Management*. La prima sezione si occupa di fornire una definizione del termine *mobile-Health*, procedendo successivamente con l'illustrazione dell'architettura di un tipico sistema che adotta questa visione. Il paragrafo si conclude con l'illustrazione dei benefici e delle sfide attualmente presenti nell'adozione di questa visione.
Il secondo paragrafo si occupa di fornire una breve panoramica sulla tematica del *self-Management*; la pratica con la quale una persona, in modo autonomo e responsabile, salvaguarda il proprio stato di salute.
Il capitolo si conclude con la presentazione di alcuni sistemi realizzati che adottano la visione del mobile Health e/o del self-Management.

2.1 Mobile Health

L'IoT consente di collegare in rete oggetti della vita quotidiana, rendendo la presenza della tecnologia sempre più pervasiva nelle nostre vite. Inoltre, negli ultimi anni, si sta assistendo a un notevole progresso tecnologico nel settore degli smartphone. Questi due fattori hanno permesso di sviluppare nell'ambito dell'healthcare il concetto di *mobile Health* (mHealth o m-Health).
Con il termine *mobile Health*, si intende la pratica con la quale si forniscono dei servizi sanitari, superando vincoli temporali e geografici, attraverso l'utilizzo di dispositivi mobili. Questa possibilità è dovuta grazie alla presenza sul mercato di device sempre più piccoli, con elevate capacità computazionali, bassi

consumi energetici e all'evoluzione delle tecnologie di comunicazione.
Il mHealth rappresenta un'opportunità sia nei paesi sviluppati che nei paesi in via di sviluppo. Nei paesi industrializzati tale pratica può essere utilizzata per migliorare i servizi sanitari già presenti; si pensi ad esempio al monitoraggio remoto degli anziani o delle malattie croniche.
Nei paesi in via di sviluppo, invece, il mHealth può essere utilizzato per fornire i servizi sanitari di base che, per diversi motivi, non possono essere forniti attraverso pratiche convenzionali. Gli ambiti in cui può essere adottato il mHealth nei paesi in via di sviluppo possono essere:

- **Educazione del paziente:** sulla base delle indagini condotte dall'organizzazione mondiale della sanità, una delle maggiori cause di morte nei paesi in via di sviluppo sono le scarse conoscenze in ambito sanitario della popolazione [4].
  Un esempio di utilizzo del mHealth, al fine di educare la popolazione per renderla consapevole sui rischi e le malattie, può essere l'utilizzo di SMS informativi. Considerando che i costi risultano essere fondamentali nei paesi in via di sviluppo, gli SMS rappresentano uno dei mezzi più economici per educare la polazione.

- **Sistemi di supporto alle decisioni:** come nei paesi industrializzati, è possibile realizzare dei sistemi di supporto alle decisioni attraverso il mHealth. Fino ad oggi, tali sistemi potevano essere utilizzati solo nei paesi industrializzati; in quanto, per la realizzazione di tali sistemi, erano necessarie delle piattaforme hardware costose. Attualmente, grazie alla potenza di calcolo che sono in grado di offrire i device mobili e, grazie all'intelligenza artificiale, tali sistemi possono essere realizzati grazie all'utilizzo di hardware a basso costo, permettendone l'utilizzo anche nei paesi in via di sviluppo

- **Addestramento del personale sanitario:** per consentire la fornitura dei servizi sanitari è fondamentale che ci sia del personale qualificato. Nei paesi in via di sviluppo il mHealth può rappresentare un ottimo strumento attraverso il quale effettuare la formazione del personale.

2.1.1 Architettura di un sistema mobile Health

Per consentire sia ai medici che ai pazienti l'accesso ai servizi in qualsiasi luogo e in qualsiasi istante, l'architettura di un tipico sistema mHealth, oltre a

prevedere l'utilizzo di device mobili, si basa fortemente sull'utilizzo della rete Internet e dei servizi web.
Le varianti con cui possono essere concepiti i sistemi mHealth sono due: *user-centric* e *provider-centric*.
Per *user-centric* si intende un tipo di servizio incentrato sul paziente. L'attore principale è l'assistito, ed è lui stesso a gestire il proprio stato di salute; accedendo a database, ai servizi sanitari nazionali e ricevendo dei feedback relativi alla propria salute.
Nei servizi *provider-centric*, invece, è colui che fornisce il servizio a visualizzare i dati del paziente, producendo delle diagnosi che vengono visualizzate dall'assistito; senza che quest'ultimo si rechi in una struttura sanitaria.
Dall'unione di queste due tipologie, è possibile realizzare dei servizi ibridi, in cui sia il paziente che il fornitore del servizio sono in grado di visualizzare i dati e, l'assistito, può prendere delle decisioni e visualizzare i feedback; anche senza l'intervento del personale sanitario.
I possibili contesti in cui realizzare sistemi provider-centric possono essere:

- Interventi di emergenza
- Monitoraggio remoto dei pazienti
- Assistenza nel processo decisionale degli operatori sanitari

Alcuni ambiti di applicazione di sistemi user-centric possono essere:

- Monitoraggio e prevenzione
- Wellness

2.1.2 Benefici del mobile Health

Di seguito i principali benefici dell'applicazione del mHealth, sia nei paesi industrializzati che nei paesi in via di sviluppo.

Rimozione di vincoli temporali e geografici

Attraverso l'utilizzo dei device mobili e delle moderne tecnologie di comunicazione, viene fornito al paziente un supporto alla mobilità.
Questo beneficio può essere riscontrato dalle persone affette da malattie croniche. Attraverso i sistemi di monitoraggio remoto, l'assistito non deve più

recarsi presso la struttura sanitaria per eseguire dei controlli periodici; in quanto l'invio dei dati da parte dei sensori, consentono all'operatore sanitario di analizzare i dati in qualsiasi momento e in ogni luogo.

Prevenzione e gestione delle malattie croniche

Le malattie croniche devono essere costantemente monitorate. Nelle strutture sanitarie vengono svolti controlli periodici per verificare lo stato della patologia. L'aggravarsi della patologia, dunque, potrebbe essere riscontrato solo al controllo successivo, ritardando quindi l'applicazione delle misure necessarie per mantenere la patologia in una situazione accettabile.
Attraverso il mHealth, è possibile realizzare sistemi di monitoraggio remoto che permettono, sia al paziente che alla struttura sanitaria, di monitorare costantemente lo stato della patologia; consentendo dunque al paziente di recarsi presso una struttura sanitaria al manifestarsi delle prime complicazioni e, agli operatori sanitari, di prendere delle decisioni che siano in grado di prevenire eventuali complicazioni e/o gestire la patologia nel migliore dei modi.

Incremento dell'accessibilità ai servizi sanitari

Come già detto, attraverso il mHealth vi è la possibilità di fornire i servizi sanitari di base nei paesi in via di sviluppo, laddove non è possibile fornirli con metodologie convenzionali.
Attraverso il mHealth, è possibile educare la popolazione e formare gli operatori sanitari.
Si possono inoltre realizzare servizi in grado di supportare le decisioni anche nei paesi poco sviluppati, dove vi sono difficoltà nel reperire ingenti quantità di fondi per finanziare la sanità.

Interventi nelle emergenze

Nei casi di emergenza, ogni secondo risulta essere fondamentale per la salute del paziente. Nella gestione delle emergenze, i fattori principali sono: anamnesi del paziente, celerità con la quale vengono fornite le prime cure e la velocità con quale viene effettuato il trasporto presso una struttura ospedaliera.
Attraverso la geolocalizzazione, di cui tutti i device mobili sono dotati, è possibile individuare il paziente e raggiungerlo nel minore tempo possibile. Inoltre, attraverso le celle cellulari, è possibile rilevare l'intensità del traffico al fine di

calcolare il percorso ottimale per recarsi dal paziente e trasportarlo presso una struttura sanitaria nel minor tempo possibile.
Infine, una volta raggiunto il paziente sul posto, il suo device mobile può essere un custode delle informazioni necessarie agli operatori sanitari (anamnesi e ultime informazioni raccolte). Attraverso questi dati, il personale sanitario può effettuare una rapida diagnosi e fornire nel più breve tempo possibile le cure di primo soccorso.

Riduzione dei costi

Grazie alla raccolta e condivisione dei dati è possibile effettuare delle diagnosi senza il bisogno che il paziente si rechi presso una struttura sanitaria, azzerando i costi da parte dell'assistito per recarsi presso la struttura, e riducendo i costi della sanità per la gestione delle strutture e dei servizi.

2.1.3 Sfide nel mobile Health

Le sfide che al momento ostacolano la realizzazione di questa visione derivano dalle tecnologie e dalle infrastrutture utilizzate. Si presentano i problemi classici dell'Internet of Things, dato che per la realizzazione di questa visione possono essere necessari sensori smart .
Inoltre, nei paesi in via di sviluppo si aggiungono delle sfide che rendono ancora più difficile l'attuazione di questa visione. Di seguito alcune delle principali sfide secondo [4].

Letteratura inadeguata nei paesi in via di sviluppo

Nei paesi in via di sviluppo, soprattutto nelle zone rurali, vi è un basso indice di alfabetizzazione. Pertanto, le conoscenze in ambito sanitario sono pressoché nulle. Ciò comporta a delle difficoltà nel comprendere le prescrizioni mediche.

Barriere linguistiche nei paesi in via di sviluppo

Come detto in precedenza, uno dei metodi attraverso il quale realizzare dei servizi di mHealth, può prevedere l'impiego di SMS. Complice una bassa alfabetizzazione e le diversità linguistiche che si possono presentare all'interno di

una nazione, si potrebbero verificare delle problematiche relative alla comprensione degli SMS da parte della popolazione, e nella comunicazione tra medico e assistito.

Mancanza di personale qualificato nei paesi in via di sviluppo

Per la realizzazione di servizi mHealth è fondamentale la presenza di personale sanitario qualificato. Nei paesi in via di sviluppo questo aspetto rappresenta attualmente una barriera nella diffusione della visione mHealth.

Mancanza di infrastrutture nei paesi in via di sviluppo

Per realizzare sistemi secondo la visione mHealth, è di fondamentale importanza la presenza di Internet. Purtroppo, tale tecnologia non è presente in tutte le aree geografiche del mondo. Allo stato attuale, la tecnologia che copre la più vasta area geografica sono gli SMS.

Costi elevati nei paesi in via di sviluppo

I costi rappresentano ancora oggi un problema per la fornitura di servizi sanitari attraverso la visione del mHealth nei paesi in via di sviluppo. Sebbene negli anni il prezzo degli smartphone entry-level si sia notevolmente abbassato, vi è ancora una buona parte della popolazione mondiale che non è in grado di acquistarli.

2.2 Self-Management

Nell'ambito dell'healthcare, per *self-Management* si intende la pratica con cui una persona prende delle decisioni al fine di salvaguardare la propria salute.
Le motivazioni che spingono ad attuare tale pratica sono diverse. Le persone affette da malattie croniche o terminali, possono scegliere di adottare alcuni accorgimenti al fine di migliorare la convivenza con la patologia o di rallentare il processo degenerativo.
Alcune persone potrebbero adottare degli accorgimenti per salvaguardare la propria salute al fine di aumentare la propria aspettativa di vita o semplicemente migliorare quest'ultima, con lo scopo di prevenire il manifestarsi di alcune patologie.

Infine, anche l'aspetto economico può essere una motivazione per la quale una persona potrebbe mettere in atto tale pratica. Cercando di seguire uno stile di vita sano, egli ridurrebbe le spese sanitarie, le quali potrebbero essere ingenti nei paesi in cui la sanità non è pubblica.
Le persone che attuano questa pratica sviluppano delle capacità di auto monitoraggio del proprio stato di salute. Affinché ciò avvenga, l'educazione che la persona riceve è fondamentale e può avvenire in diversi modi. Una delle metodologie più comuni è il rilascio da parte delle strutture sanitarie di materiale divulgativo. Inoltre, la formazione della persona può avvenire anche attraverso la dimostrazione nell'utilizzo di device medici e delle procedure per una corretta somministrazione di un farmaco.
Altro fattore determinante per la messa in pratica del self-Management, sono le informazioni di cui necessita la persona per prendere le decisioni atte a migliorare il proprio stato di salute. Con il propagarsi della visione dell'IoT e del mHealth, sono stati immessi sul mercato sistemi di supporto alle decisioni, in grado di acquisire i segnali vitali, analizzarli e fornire un feedback alla persona; al fine che quest'ultima possa effettuare la migliore scelta. Gli approcci tipicamente utilizzati per la realizzazione di questo genere di sistemi prevedono l'utilizzo di reti neurali, simulatori e analisi dei big data.

2.3 Applicazioni

Nel corso degli anni sono stati proposti in ambito accademico diverse soluzioni e, al tempo stesso, sono state messe in pratica, in forma sperimentale o in modo completamente operativo delle applicazioni di mHealth e/o self-Management. Di seguito sono presenti alcune soluzioni.

2.3.1 Self-Management Support Programme

Il programma *EDGE* (sElf-management anD support proGrammE)[5] è stato realizzato con lo scopo di consentire al paziente di effettuare il self-Management di malattie polmonari ostruttive. Si è trattato di uno studio condotto con lo scopo di comprendere l'effettiva fattibilità nella realizzazione di sistemi di mHealth finalizzati al self-Managemnt.
Il sistema prevedeva l'utilizzo da parte del paziente di un tablet Android su cui era presente un'applicazione dedicata per il self-Management della malattia. L'applicazione consentiva al paziente di:

- Tenere traccia dei sintomi
- Tenere traccia dei farmaci assunti
- Acquisire via Bluetooth, da un pulsiossimetro, la percentuale di ossigeno presente nel sangue e la frequenza cardiaca
- Visualizzare video e articoli educativi relativi alle malattie polmonari ostruttive
- Personalizzare i piani per il self-Management
- Ricevere dei messaggi da parte del personale sanitario

Per ogni paziente, i dati venivano inviati a un server remoto, il quale li elaborava attraverso un modello probabilistico realizzato su misura per ogni assistito.
Per ogni assistito, i dati raccolti nelle prime sei settimane sono stati utilizzati per determinare delle soglie di allarme, relative a: saturazione dell'ossigeno, frequenza cardiaca e ai sintomi.

2.3.2 Wireless Heart Health

Si tratta di una soluzione realizzata in Cina per consentire ai pazienti che vivono in zone rurali di effettuare uno screening cardiovascolare[6]. La soluzione realizzata prevede l'utilizzo di una smartphone dotato di sensori per effettuare l'elettrocardiogramma (ECG).
L'assistito pone il device sul proprio petto per 30 secondi circa e, al termine, una volta elaborato l'elettrocardiogramma, viene memorizzato sotto forma di Electronic Health Record e inviato attraverso l'ausilio della rete cellulare; al fine di consentire l'analisi dell'ECG da parte del personale sanitario. Quest'ultimo, dopo aver analizzato l'ECG, provvederà a contattare l'assistito attraverso una chiamata o un SMS.

2.3.3 Analisi dei carcinoma

Lo studio proponeva una soluzione che consentiva di effettuare, attraverso l'utilizzo della fotocamera di uno smartphone, l'esame per verificare la presenza di carcinomi [7].

L'applicazione consentiva di effettuare una foto con il proprio smartphone della zona di interesse. L'immagine veniva processata come un vettore a una dimensione RGB, il quale veniva inviato al server. Sul server era presente una rete neurale multi-livello feedforward, che si occupava di classificare l'immagine a essa sottoposta.
I vantaggi proposti dagli autori dello studio erano:

- Applicazione veloce e con bassi consumi energetici
- Zero costi nell'effettuare l'esame
- Possibilità di utilizzare l'applicazione in qualsiasi luogo
- Soluzione a basso costo

Lo studio non ha prodotto risultati positivi, in quanto la rete neurale non era in grado di effettuare una diagnosi corretta, generando numerosi falsi positivi e falsi negativi.

2.3.4 Prevenzione della febbre Dengue in Perù

È stata effettuata una sperimentazione in una regione del Perù con il fine di:

- Prevenire il proliferare di zanzare
- Prevenire l'esposizione a morsi di zanzare

La sperimentazione ha avuto una durata di tre mesi e, in questo arco temporale, sono stati inviati degli SMS informativi alla popolazione selezionata con lo scopo di prevenire la febbre Dengue attraverso l'educazione del cittadino [8].

2.3.5 Self-Management della pressione arteriosa

L'università del Michigan ha svolto una sperimentazione con il fine di promuovere il self-Management della pressione arteriosa in Honduras e in Messico [9]. Ai selezionati è stato fornito uno sfigmomanometro e la letteratura necessaria per comprendere il funzionamento del device.
Nel corso della sperimentazione, le persone selezionate ricevevano delle telefonate IVR attraverso le quali si ricordava agli assistiti di: controllare regolarmente la pressione, assumere i farmaci prescritti ed evitare i cibi salati. Gli

assistiti ricevevano inoltre delle mail quando almeno metà delle misurazioni riportate erano superiori o inferiori a una determinata soglia. Le segnalazioni venivano inoltre generate se gli assistiti assumevano farmaci e/o effettuavano delle misurazioni meno di due volte a settimana.

2.3.6 Controllo della tubercolosi/HIV in India

In India è stato sperimentato un sistema che fornisse assistenza agli affetti da HIV e tubercolosi [10].
L'operatore sanitario registrava il paziente al programma attraverso alcune informazioni, quali, ad esempio, la lingua con la quale preferiva ricevere le informazioni.
Ogni giorno il sistema inviava un messaggio ai pazienti, ricordandogli di assumere i farmaci. Una volta ricevuto l'SMS, i pazienti potevano rispondere, in modo predefinito, per segnalare l'assunzione dei farmaci. Una volta inviati i messaggi, il sistema scansionava le risposte.
Prima dell'invio dei messaggi, il sistema recuperava le informazioni di coloro che non avevano assunto i farmaci il giorno precedente. Tali informazioni venivano utilizzate dal personale sanitario per individuare coloro che non seguivano la cura. La prima volta che un paziente veniva segnalato dal sistema, un operatore sanitario provvedeva a contattarlo al fine di comprendere i motivi per cui non avesse assunto i farmaci; ricordandogli, inoltre, l'importanza di seguire la cura. Nel caso in cui il nome di un paziente veniva segnalato più di una volta nella lista di coloro che non assumevano i farmaci, veniva inviato presso il domicilio del paziente un operatore sanitario; al fine di comprendere le motivazioni e risolvere eventualmente il problema.
Tale studio è stato svolto in due regioni dell'India in cui vengono parlate differenti lingue. Lo scopo di questo studio era dimostrare la fattibilità nella realizzazione di un sistema per il mHealth a basso costo e senza la necessità di utilizzare Internet.

2.3.7 Reminder SMS per la cura dell'asma

In Danimarca è stato condotto uno studio finalizzato all'incremento dell'assunzione dei farmaci per la cura dell'asma attraverso l'invio di SMS [11].
Nelle prime quattro settimane è stato prescritto ai soggetti allergici un inalatore per asma. A partire della quarta settimana, alcuni di essi sono stati sele-

zionati per ricevere dei reminder attraverso degli SMS; i quali venivano inviati ogni mattina alle 10. Nel corso delle 12 settimane in cui è stata svolta la sperimentazione, si è notato un incremento del 18% nell'assunzione dell'inalatore dell'asma negli individui che ricevevano gli SMS.

2.3.8 Reminder di appuntamenti in Brasile

In Brasile è stato condotto uno studio, dalla durata di 10 mesi, in cui venivano inviati alla popolazione degli SMS per ricordare le prestazioni mediche da svolgere; con lo scopo di comprendere l'impatto di tale tecnologia sul tasso di presenza [12].
Il sistema incaricato di inviare gli SMS recuperava le informazioni da un servizio web SOAP, in cui erano presenti le informazioni relative alle prestazioni da erogare. L'SMS veniva inviato all'assistito 24 ore prima della prestazione medica, ricordandogli l'appuntamento e di confermare o meno la sua presenza. Dallo studio è emerso che, sulle cliniche su cui è stato sperimentato il sistema, il numero di appuntamenti in cui gli assistiti non si sono presentati è diminuito notevolmente.

2.3.9 Self-Management del diabete

Per un periodo di 9 mesi sono stati reclutati, da quattro stati degli USA, dei veterani di guerra affetti da diabete mellito di tipo 2 [13].
L'obiettivo dello studio era:

- Monitorare i sintomi dei pazienti e i problemi che rilevavano con il self-Management
- Fornire ai pazienti messaggi su misura relativi alla malattia e al self-Management
- Generare dei report, attraverso le mail, per gli assistenti non qualificati
- Fornire ai medici dei pazienti dei feedback relativi allo stato di salute di quest'utlimi

Gli assistiti, durante l'arco dello studio, ricevevano delle telefonate IVR. Ogni settimana veniva effettuata una chiamata per ciascun paziente. L'assistito, attraverso il tastierino numerico, rispondeva a delle domande poste dall'IVR relative alla settimana precedente. Le domande effettuate erano relative a:

- Sintomi di ipoglicemia/iperglicemia
- Test della glicemia
- Ispezione dei piedi
- Misurazioni della glicemia inferiori ai 90 mg/dl
- Misurazioni della glicemia superiore ai 300 mg/dl
- Assunzioni di farmaci per almeno due volte alla settimana per l'iperglicemia
- Trattamenti ipoglicemici
- Mancata assunzione di farmaci per l'iperglicemia
- Informazioni sul numero di misurazioni della pressione effettuate
- informazioni sulle misurazioni della pressione

Gli assistiti potevano designare un amico o un familiare come persona a cui mandare le mail riassuntive dei risultati delle chiamate svolte.
Una volta delineato un pattern sulla mancata assunzione di farmaci, sui valori anormali della glicemia e della pressione, il sistema inviava via fax un allarme al medico curante.

Capitolo 3

Il progetto: motivazioni, architettura e scelte tecnologiche

Si vuole rendere il paziente più partecipe e responsabile nel monitoraggio di una malattia cronica, quale il diabete mellito di tipo 1. Attraverso un sistema che sia in grado di fornire dei feedback all'assistito, quest'ultimo avrà la possibilità di migliorare la propria condizione di salute, migliorando la convivenza con la patologia e, al tempo stesso, riducendo le spese mediche. Pertanto, il progetto che si è deciso di realizzare, consiste in un sistema per il self-Management del diabete mellito di tipo 1, con l'obiettivo di incrementare l'empowerment del paziente attraverso un sistema user-centered.

Il capitolo è composto da un primo paragrafo in cui viene offerta una panoramica sul metabolismo del glucosio e sul diabete.
La seconda sezione presenta le caratteristiche che contraddistinguono il sistema.
Nel terzo e quarto paragrafo del capitolo sono presenti l'analisi dei requisiti e l'architettura logica che ne deriva.
Il capitolo si conclude con una panoramica sulle tecnologie selezionate per la realizzazione del sistema.

3.1 Metabolismo glucosio

All'interno del nostro organismo, i principali processi che determinano la concentranzione di glucosio nel sangue sono: la glicogenolisi, la glicogenosintesi e la digestione.
L'intestino, attraverso l'assorbimento da parte delle proprie pareti, si occupa di immettere nel flusso sanguigno il glucosio presente negli alimenti ingeriti. Al momento del pasto, attraverso il processo di assorbimento di questo organo, vi è un incremento dei livelli ematici di glucosio per un periodo di tempo compreso tra i 90 e i 120 minuti.
All'aumentare della concentrazione di glucosio presente nel flusso sanguigno, al fine di mantenerlo tra i valori normali di riferimento, il pancreas, attraverso le cellule β, secerna l'ormone dell'insulina; il cui compito è quello di regolare i livelli ematici di glucosio. La produzione di tale ormone ha un triplice ruolo:

- Stimolare l'assorbimento di glucosio da parte dei muscoli
- Stimolare il processo di glicogenosintesi da parte del fegato
- Inibire la secrezione dell'ormone del glucagone da parte delle cellule α del pancreas

Questi fattori, insieme, contribuiscono ad abbassare la glicemia. La quantità di insulina rilasciata è crescente all'aumentare della concentrazione di glucosio presente nel sangue. Quando quest'ultima è inferiore ai 59 mg/dl, le cellule β cessano la secrezione di insulina.
Il fegato è in grado di assorbire il 60% circa di glucosio, mentre il restante 40% viene assorbito dai muscoli. L'assorbimento del glucosio avviene attraverso il processo di glicogenosintesi. Si tratta di un processo che consente di convertire il glucosio in glicogeno. Il glicogeno è un polimero del glucosio e funge da riserva energetica qualora quest'ultima sia richiesta. I muscoli e il fegato fungono da siti di stoccaggio di questo polimero.
Le cellule α si occupano di secernere il glucagone. Tale ormone ha un ruolo diametralmente opposto a quello dell'insulina. Quando rilasciato, anziché stimolare l'assorbimento del glucosio, stimola il rilascio di quest'ultimo da parte del fegato, mantenendo il valore basale nei valori di riferimento. Quando la concentrazione di glucosio è inferiore al livello di riferimento, la produzione di glucagone, aumenta in modo crescente al diminuire della glicemia. Come intuibile, durante e in prossimità dei pasti, l'attività delle cellule α è piuttosto

bassa.
Quando è in corso un'attività fisica, i muscoli utilizzano il glicogeno immagazzinato e, una volta esauritosi, procedono con l'assorbimento del glucosio presente nel flusso sanguigno. Come detto poc'anzi, al diminuire del livello ematico di glucosio, viene rilasciato l'ormone del glucagone. Tale ormone stimola la produzione di glucosio da parte del fegato, il quale avvia il processo di glicogenolisi; un processo metabolico in grado di trasformare le molecole di glicogeno in glucosio.
Gli ormoni dell'insulina e del glucagone, attraverso la loro produzione, aiutano l'organismo a mantenere la concentrazione di glucosio nei livelli di riferimento.

3.1.1 Diabete mellito

Il diabete è una patologia del metabolismo che comporta delle difficoltà nella secrezione o sull'azione dell'insulina, causando uno stato di iperglicemia. Un'elevata concentrazione di glucosio, per lunghi periodi di tempo, può portare a numerose complicazioni. Le principali patologie sono due: diabete mellito di tipo 1 e 2.

Diabete mellito di tipo 1

Il diabete mellito di tipo 1 consiste nell'autodistruzione delle cellule β da parte dell'organismo. Questo processo è dovuto a delle predisposizioni genetiche e, a volte, correlate da fattori ambientali non ancora ben definiti. Trattandosi di predisposizioni genetiche, la maggior parte dei casi si presenta in età infantile/adoloscenziale; al punto da essere definito anche come diabete giovanile.
Poiché l'organismo non è in grado di produrre l'ormone dell'insulina, al fine di mantenere la concentrazione di glucosio nei livelli di riferimento, sono necessarie delle iniezioni di insulina.
La terapia tipicamente adottata prevede la somministrazione di due tipologie di insulina. Una ad azione lenta, la quale rilascia lentamente dell'insulina in modo costante per 24 ore. La seconda tipologia è tipicamente ad azione rapida o ultrarapida, da iniettare in concomitanza dei pasti. Quest'ultima rilascia dell'insulina per brevi periodi di tempo al fine di far fronte ai picchi glicemici post-prandiali.

Degli accorgimenti sull'alimentazione e sull'attività fisca svolta possono aiutare a ridurre i boli di insulina da iniettare.

Diabete mellito di tipo 2

Questo tipo di patologia è dovuta a un'insulino resistenza da parte dell'organismo. A differenza del diabete mellito di tipo 1, le cellule β sono in grado di secernere l'insulina, ma l'organismo sviluppa una sorta di resistenza a quest'ultima. Per insulino resistenza si intende una minore capacità dell'organismo nell'assorbire il glucosio presente nel flusso sanguigno in base alla quantità di insulina presente.
Questa particolare patologia si sviluppa nella tarda età, al punto da essere denominata anche diabete degli anziani. Altre cause che comportano un'insulino resistenza da parte dell'organismo possono essere l'obesità e la mancanza di attività fisica.
Per questo tipo di patologia, in base alla gravità, per mantenere la concentrazione di glucosio all'interno dei valori di riferimento, possono essere necessari degli accorgimenti sull'attività fisica e sull'alimentazione. Nei casi più gravi, può essere necessaria la somministrazione di farmaci; ma senza giungere a una terapia che prevede delle iniezioni di insulina.

3.2 Caratteristiche del sistema

Sulla base delle motivazioni enunciate nell'introduzione del capitolo, si vuole realizzare un sistema in cui il paziente sia in grado, attraverso il proprio smartphone, di:

- Importare automaticamente le misurazioni relative alla pressione arteriosa da uno sfigmomanometro
- Importare automaticamente le misurazioni relative alla concentrazione di glucosio da un glucometro
- Visualizzare l'elenco delle misurazioni relative alla pressione arteriosa o alla glicemia
- Simulare l'andamento glicemico real time. Durante lo svolgimento della simulazione, il paziente dovrà essere in grado di inserire i pasti che effet-

tuerà. Dovrà inoltre essere possibile acquisire, in modo automatico, la frequenza cardiaca del paziente

- Effettuare una simulazione del proprio andamento glicemico nell'arco di una settimana. Il paziente dovrà essere in grado di inserire i pasti che effettuerà e le diverse attività fisiche che svolgerà
- Visualizzare dei suggerimenti sullo stile di vita da seguire ogni qualvolta che termina una simulazione o una sessione real time. I suggerimenti dovranno essere prodotti da un sistema di regole definito dal medico curante

Per quanto concerne i primi due punti, l'applicazione dovrà essere in grado di acquisire le misurazioni presenti in memoria da uno sfigmomanometro e da un glucometro, attraverso i quali, il paziente monitorerà il proprio stato di salute. L'import delle misurazioni da parte del dispositivo mobile dovrà avvenire su specifica richiesta del paziente.
Per quanto concerne il quarto punto, tale funzionalità dovrà consentire al paziente di monitorare il possibile valore glicemico, evitando misurazioni invasive. Ad ogni minuto il paziente dovrà visualizzare sul proprio dispositivo i seguenti dati:

- Concentrazione di glucosio
- Calorie bruciate dal momento in cui è iniziata la sessione
- Frequenza cardiaca

La frequenza cardiaca dovrà essere acquisita automaticamente dal dispositivo mobile. Inoltre, il paziente, nel corso della simulazione, dovrà essere in grado di inserire i pasti che effettuerà e dovrà visualizzare il numero di boli di insulina da iniettare. I dati prodotti dalla simulazione dovranno essere inviati a un server; al fine di permettere al medico di osservare le sessioni effettuate dal paziente.
Per quanto concerne il quinto punto, tale funzionalità dovrà consentire al paziente di simulare l'andamento glicemico settimanale. Prima di avviare la simulazione, il paziente dovrà inserire i pasti e le attività fisiche che intende simulare. Per le attività fisiche dovrà specificare: orario di inizio, orario di fine e calorie bruciate al minuto. I dati prodotti dalla simulazione dovranno

essere inviati a un server, al fine di permettere al medico di osservare le sessioni effettuate dall'assistito.

Il sistema dovrà consentire a un medico, attraverso un'interfaccia web, di:

- Visualizzare l'elenco dei pazienti in cura
- Visualizzare per ogni singolo paziente le misurazioni relative alla pressione arteriosa e alla glicemia
- Visualizzare l'elenco delle simulazioni e delle sessioni real time effettuate da un particolare paziente
- Visualizzare per una particolare sessione o simulazione il dettaglio di quest'ultima.
- Inserire, rimuovere o modificare una regola per un dato paziente

Il sistema di regole dovrà analizzare i dati relativi all'andamento glicemico del paziente; al fine di produrre dei suggerimenti per migliorare lo stile di vita di quest'ultimo. Ogni suggerimento sarà formato da una coppia di valori: il regime alimentare da seguire (basso, medio ed elevato) abbinato con l'intensità dell'attività fisica da svolgere (basso, medio ed elevato).
Il medico curante potrà definire, per ciascun paziente in cura, delle regole che restituiranno un suggerimento riguardo lo stile di vita da seguire.
Il medico curante potrà definire le condizioni in cui una determinata regola dovrà produrre un feedback. Esso potrà definire le condizioni relative a:

- Picco glicemico medio post prandiale
- Coefficiente angolare medio tra i picchi glicemici post prandiali e il valore glicemico al momento del pasto
- Coefficiente angolare medio tra i picchi glicemici post prandiali e il minimo valore glicemico registrato tra le due e quattro ore successive al pasto
- Carico glicemico medio della giornata (basso, medio, elevato)
- Intensità attività fisica svolta durante la giornata (basso, medio, elevato)

Oltre a definire le condizioni in cui la regola dovrà fornire un suggerimento, il medico potrà definire anche i vincoli che il suggerimento dovrà soddisfare. In particolare potrà definire:

- Il carico glicemico medio giornaliero
- L'intensità dell'attività fisica giornaliera

La regola, quando determinerà i suggerimenti da restituire al paziente, oltre a tenere conto dei vincoli definiti dal medico, dovrà considerare anche le preferenze espresse dal paziente. Pertanto, l'assistito potrà indicare, sia per l'alimentazione che per l'attività fisica, il livello che preferirà seguire.
Le preferenze espresse dal paziente potrebbero entrare in conflitto con i vincoli definiti dal medico. In tal caso il sistema non sarà in grado di produrre una soluzione ammissibile. Al fine di garantire almeno una soluzione ammissibile, il sistema dovrà effettuare, se necessario, un rilassamento sui vincoli espressi dall'assistito.

3.3 Analisi dei requisiti

Al fine di evitare ambiguità e comprendere in modo chiaro i requisiti, è utile definire un glossario.

Termine	**Definizione**
Self-Management	Pratica con cui una persona monitora il proprio stato di salute autonomamente e attua alcune decisioni per migliorarlo
Sfigmomanometro	Dispositivo che consente di misurare la pressione arteriosa
Glucometro	Dispositivo che consente di misurare la concentrazione di glucosio presente nel sangue
Sessione real time	Lasso di tempo in cui il paziente effettua un monitoring real time della glicemia

Regola	Sistema che analizza i dati prodotti dalla sessione real time e/o dalla simulazione e fornisce un feedback al paziente
Boli di insulina	Quantità di insulina da iniettare in concomitanza dei pasti
Soluzione ammissibile	Soluzione che soddisfa tutti i vincoli espressi dal medico e dal paziente
Rilassamento dei vincoli	Tecnica che estende il numero di soluzioni ammissibili

3.3.1 User stories

Le user stories permettono di definire in modo sintetico le funzionalità che il committente chiede che vengano implementate. Non contengono dettagli tecnici, ma semplicemente i bisogni che riguardano le caratteristiche del prodotto richiesto. Sono espresse attraverso il linguaggio naturale, in modo che possano essere compresi sia dagli addetti ai lavori che dal committente.
Di seguito sono presenti le user stories relative al paziente e al medico; le quali ci permetteranno in seguito di definire l'architettura del sistema.

- Come paziente vorrei importare i dati relativi alla pressione arteriosa dallo sfigmomanometro al dispositivo mobile in modo automatico
- Come paziente vorrei importare i dati relativi alla glicemia dal glucometro al dispositivo mobile in modo automatico
- Come paziente vorrei effettuare una simulazione real time dell'andamento glicemico
- Come paziente vorrei simulare l'andamento glicemico nell'arco di una settimana
- Come paziente vorrei ricevere dei consigli che tengono conto delle mie preferenze su come correggere lo stile di vita
- Come medico vorrei visualizzare l'elenco dei pazienti in cura

- Come medico vorrei visualizzare, per un dato paziente, le misurazioni da esso effettuate relative alla pressione arteriosa

- Come medico vorrei visualizzare, per un dato paziente, le misurazioni da esso effettuate relative alla glicemia

- Come medico vorrei visualizzare l'elenco delle simulazioni effettuate da un dato paziente

- Come medico vorrei visualizzare l'elenco delle sessioni real time effettuate da un dato paziente

- Come medico vorrei visualizzare il dettaglio relativo a una simulazione effettuata da un dato paziente

- Come medico vorrei visualizzare il dettaglio relativo a una sessione real time effettuata da un dato paziente

- Come medico vorrei inviare delle notifiche a un dato paziente

- Come medico vorrei aggiungere, modificare, ed eliminare delle regole per un dato paziente

3.4 Architettura logica

In questa fase non verranno presi in considerazione nè i dettagli implementativi nè le tecnologie da utilizzare; in quanto è fondamentale che l'architettura del sistema sia technology independent.
In base all'analisi dei requisiti, il sistema si può suddividere nei seguenti sottosistemi: *body gateway, server, applicazione mobile e body area network*. La componente *applicazione mobile*, può essere ulteriormente suddivisa in: *modello diabete* e *regole*.

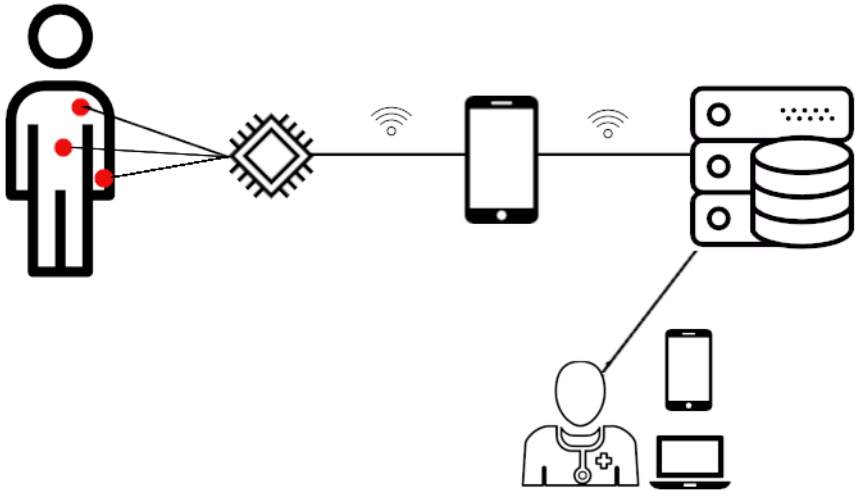

Figura 3.1: Architettura logica del sistema

Come si evince dall'immagine, per la realizzazione del sistema, sarà presente una body area network; la quale, attraverso i sensori che andranno a comporla, acquisirà alcuni parametri vitali del paziente. I sensori che costituiranno la body area network saranno: sfigmomanometro, pulsiossimetro e glucometro.
Per evitare che il device mobile del paziente debba gestire la comunicazione con i sensori della body area network e, per garantire un'adeguata scalabilità, nel caso in cui il numero di sensori dovesse aumentare, l'acquisizione e l'invio dei dati verrà affidata al body gateway; il quale, una volta stabilita la connessione con lo smartphone del paziente, potrà comunicare con esso. In particolare, il device invierà delle richieste di acquisizione dati e, per ognuna di essa, il body gateway recupererà le informazioni dalla body area network.
L'applicazione invierà al server le misurazioni acquisite, i dati prodotti dalle simulazioni e dalle sessioni real time effettuate dal paziente. Al momento del login da parte dell'assistito, inoltre, l'applicazione recupererà le regole definite dal medico; le quali verranno utilizzate da parte dell'applicazione per la produzione dei feedback.
Il server si occuperà di fornire l'interfaccia web per il medico, attraverso la quale potrà visualizzare le informazioni relative ai propri pazienti, inviare notifiche e definire le regole per ognuno di essi. Inoltre, come detto poc'anzi, si

occuperà di memorizzare tutte le informazioni prodotte dal paziente e, fornirà all'applicazione di quest'ultimo, l'insieme di regole definito dal medico curante. L'architettura del sistema che emerge dall'analisi dei requisiti, può essere rappresentata attraverso un diagramma di struttura (logica); al fine di illustrare in modo completo le componenti che compongono il sistema. Per ogni componente del sistema, il software può essere suddiviso nelle seguenti dimensioni: *informazione*, *controllo* e *presentazione*.

- **Server:** componente dell'architettura in cui risiederanno i dati dei pazienti e le logiche per la generazione delle regole che si occuperanno di fornire un feedback a quest'ultimo. Inoltre, tale componente si occuperà di fornire l'interfaccia web attraverso la quale il medico potrà visualizzare le informazioni relative agli assistiti in cura.
  Lo strato di presentazione sarà costituito esclusivamente dalle pagine HTML che andranno a comporre l'interfaccia web.
  Lo strato di controllo comprenderà tutti i moduli che verranno utilizzati per l'interazione con il medico e con il paziente.
  Lo strato di informazione sarà costituito dall'insieme delle entità rappresentanti i dati del dominio applicativo.

- **Applicazione mobile:** l'applicazione per smartphone rappresenterà il cuore del sistema. Il paziente, attraverso di essa, potrà simulare l'andamento glicemico settimanale e, inoltre, potrà visualizzare in tempo reale una simulazione della propria glicemia, evitando misurazioni invasive.
  Il livello di presentazione sarò costituito dalla user interface dell'applicazione.
  Il livello di controllo sarà costituito da cinque controller. Il primo controller si occuperà di gestire le misurazioni relative alla pressione arteriosa e alla glicemia. Il secondo controller si occuperà di gestire le simulazioni del paziente. Il terzo controller si occuperà di gestire le sessioni real time. Il quarto controller si occuperà di gestire i feedback prodotti dal sistema di regole. Infine, l'ultimo controller, si occuperà di stabilire una connessione con il body gateway.
  Il livello di informazione sarà costituito dall'insieme dei feedback prodotti dal sistema di regole, dall'insieme delle misurazioni della pressione arteriosa, dall'insieme delle misurazioni della glicemia e dall'insieme dei

dati prodotti dal modello del diabete nel corso delle simulazioni e delle sessioni real time.
Come detto poc'anzi, questa componente può essere suddivisa in due sottocomponenti: *modello diabete* e *regole*.

- **Modello diabete**: questa componente si occuperà di simulare il metabolismo del glucosio di una persona diabetica ogni qualvolta che l'assistito richiederà una simulazione o una sessione real time.
  Il livello di controllo sarà costituito dai controller, descritti poc'anzi, relativi alla simulazione e alla sessione real time.
  Il livello di informazione sarà costituito dalle informazioni prodotte dal modello nel corso della simulazione del metabolismo del glucosio.
- **Regole**: questa componente si occuperà di applicare le regole definite dal medico curante per la produzione dei feedback relativi allo stile di vita consigliato.
  Il livello di controllo sarà costituito dal controller, descritto poc'anzi, relativo alla gestione dei feedback.
  Il livello di informazione sarà costituito dai risultati prodotti in seguito all'analisi delle regole.

- **Body gateway:** sebbene tale componente non emerga in modo esplicito nei requisiti, è necessaria ai fini realizzativi del sistema; in quanto si occuperà di acquisire e inviare all'applicazione i dati provenienti dalla body area network.
  Sarà presente soltanto il controllo in questa componente; il cui compito sarà quello di stabilire una comunicazione con l'applicazione e di inviare i dati provenienti dalla body area network.

- **Body Area Network:** tale componente interagirà solo con il body gateway, il quale recupererà le informazioni dai sensori che compongono la body area network.
  Il livello di controllo si occuperà di ricevere le richieste di acquisizione dati provenienti dal body gateway e di fornire a quest'ultimo le misurazioni dei sensori che compongono la body area network.
  Il livello di informazione sarà costituito semplicemente dai dati biomedici dell'assistito provenienti dai sensori.

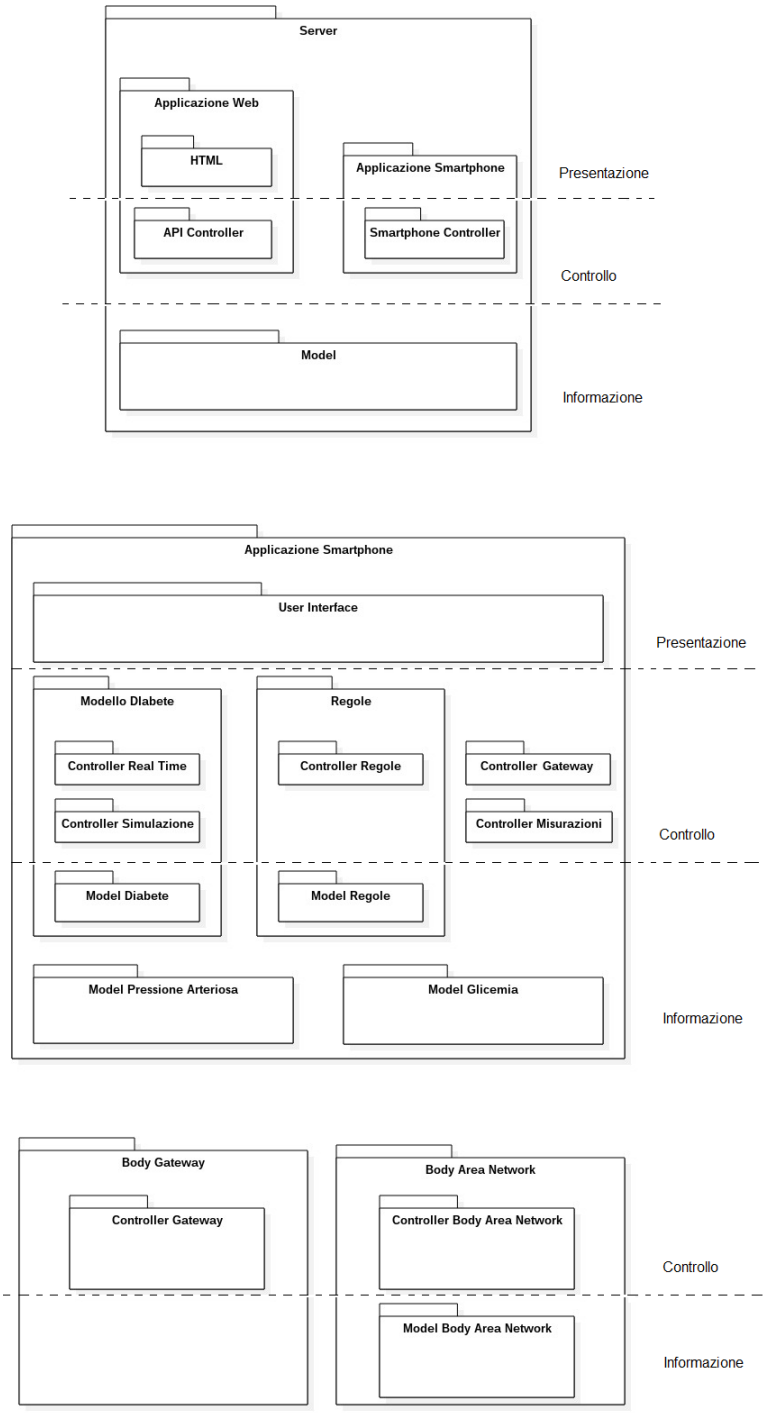

Figura 3.2: Architettura logica del sistema suddivisa nelle dimensioni di: presentazione, controllo e informazione

3.4.1 Interazione tra le componenti

Di seguito è possibile notare, attraverso i diagrammi di sequenza, l'interazione tra le varie componenti.
Prima di procedere con l'illustrazione dell'interazione tra le componenti, si tiene a precisare che, in base all'importanza della componente per una particolare interazione, quest'ultima verrà visualizzata con un diverso livello di dettaglio.
Nella figura successiva è possibile notare le possibili interazioni tra lo smartphone e il body gateway.

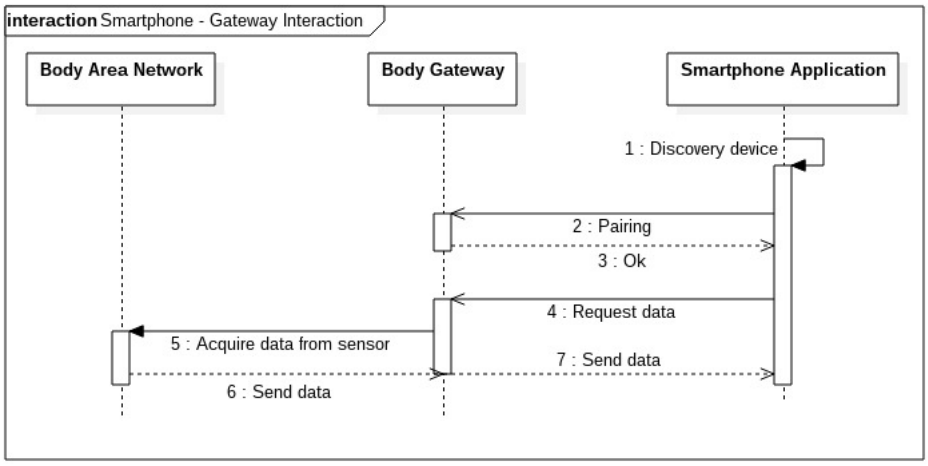

Figura 3.3: Diagramma di sequenza connessione

Come visibile dalla figura, la prima interazione tra le due componenti consiste nel stabilire una connessione. Poiché il gateway non può conoscere il device con cui stabilire la connessione, sarà lo smartphone a inviare la richiesta.
Stabilita la connessione, vi possono essere tre tipi di interazione, tutte finalizzate al recupero dei dati. In ogni interazione, sarà lo smartphone ad inviare una richiesta di acquisizione dati, specificando il tipo di dato a cui è interessato. Il gateway, una volta ricevuta la richiesta, recupererà i dati dalla body area network e li invierà allo smartphone.

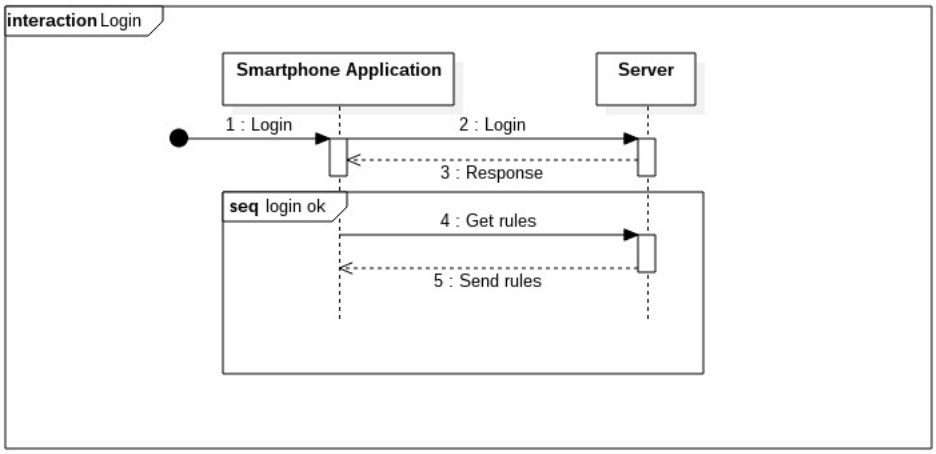

Figura 3.4: Diagramma di sequenza login

Il diagramma di sequenza presente in figura 3.4, descrive l'interazione tra le varie parti del sistema al momento del login. Al momento dell'inserimento delle credenziali da parte dell'utente, l'applicazione invierà una richiesta HTTP POST al server; Il quale controllerà che le credenziali inserite siano corrette. In caso di esito positivo, l'applicazione, oltre a consentire l'accesso all'utente, invierà una richiesta al server; attraverso la quale chiederà le regole definite dal medico. Se presenti, il server le invierà all'applicazione, altrimenti invierà una risposta senza alcun payload.
Il diagramma di sequenza presente in figura 3.5, illustra l'interazione che avviene tra le varie componenti del sistema al momento dell'acquisizione delle misurazioni presenti nello sfigmomanometro e/o glucometro.
Al momento dell'acquisizione delle misurazioni da parte del paziente, l'applicazione invierà una richiesta al body gateway, il quale restituirà le misurazioni di interesse. Alla ricezione delle misurazioni, l'applicazione, le invierà al server; il quale le memorizzerà e restituirà tutte le misurazioni effettuate dal paziente. Si è scelto di definire questo tipo di comportamento in quanto, il paziente, potrebbe utilizzare più di un device su cui effettuare le simulazioni o visualizzare le misurazioni. Adottando questa soluzione, si crea una forma di sincronizzazione con il server e, nel caso in cui l'assistito dovesse cambiare device, avrà a disposizione l'elenco completo delle misurazioni effettuate.

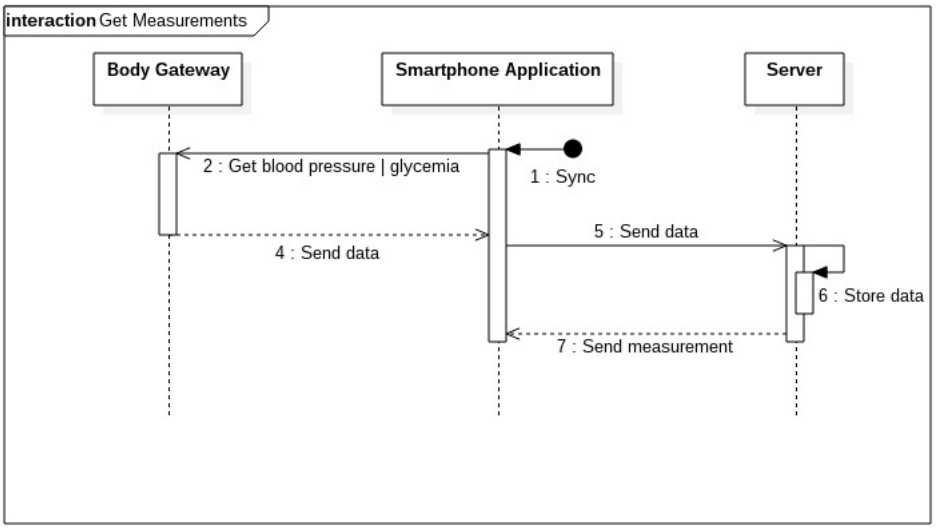

Figura 3.5: Diagramma di sequenza acquisizione misurazioni

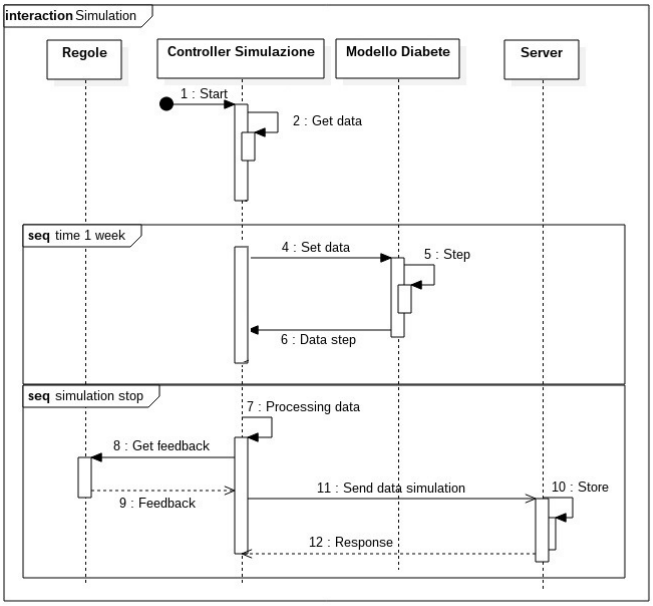

Figura 3.6: Diagramma di sequenza simulazione

Il diagramma di sequenza presente in figura 3.6, mostra l'interazione tra le varie componenti nel caso della simulazione settimanale.
Al momento dell'avvio della simulazione da parte dell'assistito, il controller si occuperà di passare i parametri definiti dal paziente al modello; al fine di configurare l'ambiente e avviare la simulazione.
Nel corso della simulazione, Il modello invierà, ad ogni step, i dati prodotti al controller. Quest'ultimo, oltre a ricevere i dati, invierà al modello i dati da utilizzare per lo step successivo. Al termine della simulazione, il controller effettuerà un processing dei dati e, invierà quest'ultimi, al sistema di regole; il quale produrrà un feedback che verrà restituito al controller. Ottenuto il feedback, il controller mostrerà i risultati al paziente e invierà i dati della simulazione al server.

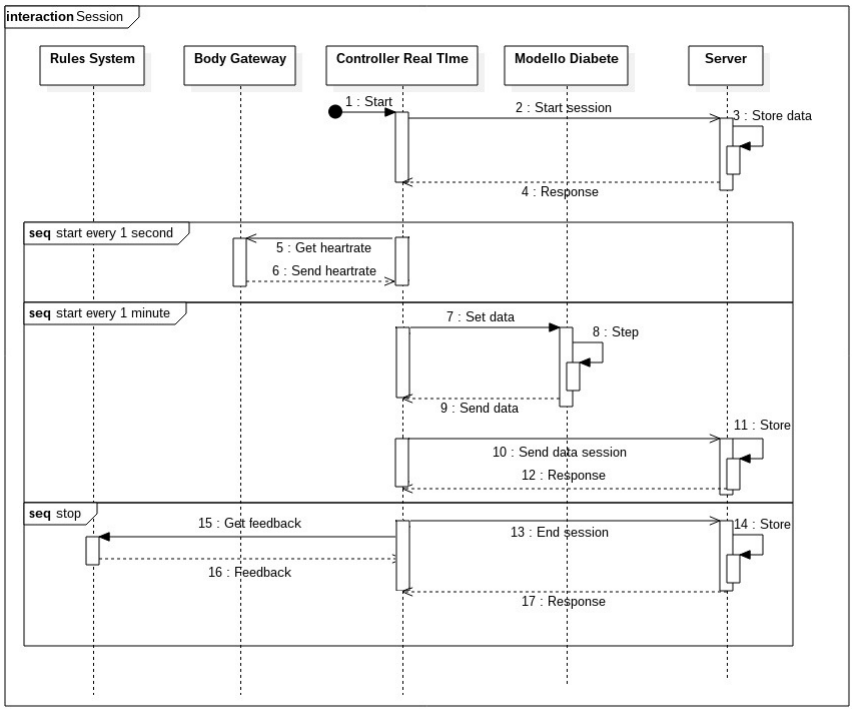

Figura 3.7: Diagramma di sequenza sessione real time

L'interazione tra le componenti nel caso di una sessione real time è pres-

soché identica al caso della simulazione, con solo due differenze. Come visibile dal diagramma, è presente la componente gateway, la quale verrà coinvolta quando il controller chiederà informazioni sulla frequenza cardiaca. La seconda differenza, consiste nell'invio dei dati al server. A differenza della simulazione, i dati verranno inviati al server ogni minuto e non al termine della sessione.

3.5 Tecnologie utilizzate

3.5.1 Bluetooth

Per consentire la comunicazione tra lo smartphone dell'assistito e il body gateway, al fine di acquisire i dati proveniente dalla body area network, è nostra intenzione utilizzare la tecnologia Bluetooth. Tale tecnologia risulta essere ormai diffusa su qualsiasi dispositivo mobile e, pertanto, è possibile garantire la più ampia copertura per quanto concerne di device utilizzabili dall'assistito. La tecnologia Bluetooth, inoltre, offre un ottimo compromesso tra raggio di copertura e velocità di trasmissione; consentendo un rapido trasferimento dei dati e, permettendo all'assistito di non dover tenere il proprio device mobile in prossimità del body gateway.

3.5.2 HTTP

Per consentire l'interazione tra il medico e il server, attraverso l'applicazione web, la scelta di utilizzare HTTP risulta essere ovvia, ma è nostra intenzione utilizzare questo protocollo di comunicazione anche per l'interazione tra il server e lo smartphone dell'assistito. Come si evince dai diagrammi di sequenza illustrati precedentemente, la comunicazione tra il server e il device dell'assistito, presenta gli aspetti tipici della comunicazione client-server. Pertanto, l'adozione di HTTP risulta essere la scelta migliore per l'interazione tra queste componenti.

3.5.3 Android

Si è scelto di realizzare un'applicazione nativa per la piattaforma Android. La scelta è legata dalle tecnologie necessarie per la realizzazione del sistema. Poiché bisogna simulare e prevedere l'andamento glicemico del paziente, è necessario realizzare un modello attraverso il quale effettuare la simulazione.

Poiché per il modello, si è scelto di adoperare un framework Java, si evince che l'unica scelta possibile sia la realizzazione di un'applicazione nativa Android.

3.5.4 Java

Per la realizzazione delle funzionalità relative al body gateway si è scelto di utilizzare Java per tre motivi.
Uno dei motivi per cui si è scelto di utilizzare Java è la capacità di fornire un livello di astrazione più elevato; ciò consente allo sviluppatore di non dover programmare a basso livello.
Il secondo motivo per il quale è stata scelta questa tecnologia è legata a un aspetto pratico: l'esistenza di una libreria per accedere ai dati della piattaforma utilizzata per realizzare la body area network.
Infine, uno dei motivi per cui è stata scelta tale tecnologia, è data dalla portabilità che offre; in quanto il software prodotto potrà essere eseguito su piattaforme hardware e software eterogenee.
Sebbene uno dei requisiti nell'IoT, soprattutto nell'ambito medico, è la velocità con cui il sistema trasmette le informazioni, la scelta di tale soluzione può essere controproducente. Poiché l'ambito per cui verrà realizzato il sistema non è sottoposto a vincoli hard-real time e, un eventuale ritardo nella comunicazione dei dati non può danneggiare il paziente, si è scelto comunque l'adozione di tale tecnologia.

3.5.5 NodeJs

Per la realizzazione della parte server si è scelto di adoperare NodeJs; un ambiente Javascript che utilizza il motore V8 di Google. Sia il motore V8 che NodeJs puntano su un basso utilizzo della memoria e su alte prestazioni.
A differenza dei vari ambienti di sviluppo lato server, NodeJs non fa affidamento sul multithreading per eseguire in modo concorrente la business logic, ma si basa sugli eventi di I/O asincroni. NodeJs si basa su un modello event-driven, il quale consente l'utilizzo di un solo thread per la gestione dell'intera business logic, da cui deriva il basso consumo di memoria. La programmazione a eventi è supportata in modo nativo.
I moderni web-server prevedono molte operazioni di I/O bloccanti e, al fine di gestire più richieste in concorrenza, creano dei thread. Ciò pone sotto stress

la macchina su cui è in esecuzione il server. Inoltre, i thread sono difficili da gestire, in quanto si possono verificare casi di deadlock e/o di starvation. In NodeJs, ogni operazione di I/O è asincrona e sono gestite tramite funzioni high-order; ovvero attraverso funzioni che accettano come parametri altre funzioni. Le funzioni passate come parametro rappresentano le istruzioni da eseguire al verificarsi dell'evento.
Il flusso del programma è determinato dalle funzioni che sono esplicitamente chiamate. Queste funzioni non sono mai bloccanti sulle operazioni di I/O, ma registrano delle callback da chiamare al termine.
L'event loop che gestisce gli eventi è integrato in NodeJs. Lo sviluppatore non deve fare altro che definire gli handler; rendendo più semplice la gestione della concorrenza.
I vantaggi di NodeJs si possono riassumere in:

- Alte prestazioni
- Basso consumo di memoria
- Facile realizzazione di servizi web concorrenti
- Utilizzo della stessa tecnologia sia lato client che lato server
- Modularità
- Open-source

NodeJs è open source e inoltre è modulare. Attraverso il packet manager *npm*, è possibile installare i moduli necessari per la realizzazione del nostro programma in NodeJs. Un modulo è l'equivalente di una libreria.
I motivi per cui si è scelto di utilizzare NodeJs per la realizzazione del server è dovuto alla possibilità di gestire un alto numero di richieste in concorrenza, mantenendo comunque un uso delle risorse basso. Considerando che il numero di richieste che possono giungere in un contesto del genere è piuttosto elevato, è necessario realizzare un server che sia in grado di scalare e accettare il maggior numero di richieste possibili in concorrenza. Il secondo motivo per cui si è scelto di utilizzare NodeJs è dovuto dalla possibilità di utilizzare lo stesso linguaggio sia lato server che lato client.

3.5.6 Prolog

Prolog è un linguaggio di programmazione basato sul paradigma della logica del primo ordine. Tale paradigma consente di definire la struttura logica di un problema, piuttosto che una soluzione, consentendo allo sviluppatore di non focalizzarsi sui dettagli che consentono la risoluzione del problema.
Tale tecnologia è stata scelta per la realizzazione del sistema di regole; In quanto il medico, attraverso le regole che definisce, determina la struttura logica necessaria per elaborare un feedback.

Sintassi

La logica del programma è espressa attraverso delle relazioni e, la risoluzione dei problemi avviene attraverso l'interrogazione di quest'ultime.
La sintassi Prolog è costituita da:

- Termini
- Regole
- Fatti

I termini possono essere delle costanti, variabili o termini composti. Le costanti possono essere rappresentati da numeri, stringhe e atomi (2.5, 2, 'Hello World', hello).
Le variabili sono rappresentate da una stringa alfanumerica e il primo carattere può essere underscore o una lettera maiuscola.
I termini composti, detti anche funtori, sono formati da uno o più termini (date(17,'May',2018)).
Un fatto ha la seguente sintassi:

nome_fatto(termine1,...,termineN).

La peculiarità dei fatti è che sono considerati sempre veri. Non sono da confondere con i termini. I fatti, al termine presentano un punto, mentre i termini sono una porzione di dati utilizzata all'interno dei fatti o delle regole.
Le regole hanno la seguente sintassi:

testa(termine1,...,termineN):- corpo.

La regola viene letta nel seguente modo: testa è vera se corpo è vero. Attraverso di esse stabiliamo che un predicato è vero, ammesso che gli altri predicati siano veri.
Di seguito è possibile trovare un esempio di programma prolog con i costrutti appena illustrati.

```
1 parents(dave,jude,johnny).
2 parents(bob, megan, fiona).
3 parents(philip, debby, jude).
4 parents(nick, allison, megan).
5
6 grandparents(X,Y,Z):-
7   parents(X,Y,W),
8   (parents(W,S,Z);parents(S,W,Z)).
```

Il programma definisce dei fatti che rappresentano la relazione: padre, madre e figlio. Johhny è figlio di Dave e Jude, Fiona è figlia di Bob e Megan, Jude è figlia di Philip e Debby, Megan è figlia di Nick e Allison.
La regola *grandparents\3*, consente di stabilire la relazione nonni-nipoti. X e Y sono le variabili che rappresentano i nonni, mentre Z il nipote. Per affermare che Z è nipote di X e Y, è necessario che W sia figlio di X/Y e che Z sia figlio di W/S oppure di S/W. Per la risoluzione della regola *grandparents*, si tenta di risolvere i predicati del corpo. Tra un predicato e l'altro, il separatore "," rappresenta un and, mentre il ";" rappresenta un or. Pertanto, affinché Z sia nipote di X/Y, quest'ultimi dovranno essere genitori di W, mentre Z dovrà essere figlio di W/S o di S/W. Se si effettua la seguente query:

grandparents(philib,debby,johnny).

Prolog restituirà true. Se al posto di *johnny*, ci fosse stata *fiona*, il risultato restituito sarebbe stato false.
Prolog non consente solo di verificare se una regola è vera o meno, ma consente anche di ottenere tutti i possibili risultati che soddisfano una regola. Ad esempio, se si effettua la query:

grandparents(X,Y,Z).

Prolog restituirà:

- grandparents(philip , debby, johnny)
- grandparents(nick, allison, fiona)

È possibile effettuare anche una query del tipo:

grandparents(X,Y,johnny).

La quale chiede chi sono i nonni di johnny.

3.5.7 RethinkDb

Per la memorizzazione dei dati provenienti dal paziente e delle regole definite dal medico curante, si è scelto di utilizzare un database NoSQL (Not Only SQL). La peculiarità di questo tipo di database è quella di non utilizzare un modello relazionale.
Per la memorizzazione dei dati, tra i vari database NoSQL, si è scelto di utilizzare *RethinkDB*. Si tratta di un database orientato ai documenti, in cui i dati non vengono memorizzati in tabelle con campi uniformi per ogni record, ma ogni record è memorizzato come un documento con determinate caratteristiche.
RethinkDB è un database NoSQL open-source, scalabile, concepito per essere utilizzato con applicazioni real-time. La tecnologia con la quale vengono memorizzati i documenti si basa su JSON. La caratteristica principale di questo database è che per accedere ai dati, soprattutto nei casi in cui questi provengano in tempo reale, è possibile usufruire del pattern publish-subscribe; grazie al quale si viene notificati ogni qualvolta che vengono aggiunti e/o modificati dei dati.
RethinkDB risulta essere un'ottima scelta quando la soluzione che si intende realizzare, necessita di dati in tempo reale. Nel nostro caso di studio, poiché un medico potrebbe voler osservare lo stato del paziente in tempo reale, risulta essere la migliore scelta.
Poiché i sistemi IoT producono una mole di dati non trascurabile, l'utilizzo dei più tradizionali database relazioni risulta non essere adatto; soprattutto a causa dei tempi necessari per recuperare le informazioni. Per questo motivo si è scelto un database non relazione per la memorizzazione dei dati prodotti.

3.5.8 Angular

Angular è un framework che consente la realizzazione di Single Page Application; ovvero delle applicazioni web costituiti da una sola pagina; con lo scopo

di fornire una user experience più piacevole per l'utente, evitando le interruzioni per le richieste di pagine al server.
Angular permette di realizzare le applicazioni attraverso l'ausilio di Typescript; un superset tipizzato che compila codice Javascript.

Architettura

L'architettura di Angular è basata sul concetto dei moduli, detti *ngModules*; i quali forniscono un contesto di compilazione per i *components*. Ogni applicazione Angular presenta almeno un modulo base, di root, detto *AppModule*. Il modulo di root fornisce il meccanismo di bootstrap per il lancio dell'applicazione.
I moduli possono essere importati e possono esporre delle funzioni; al fine di permettere l'utilizzo di quest'ultimi da parte dei moduli che li importano.
L'organizzazione in moduli, similmente al concetto di classi nella programmazione ad oggetti, permette uno sviluppo modulare, favorendone la manutenibilità e la riusabilità.

Oltre ai moduli, l'architettura di Angular prevede i *Components*. Ogni applicazione, come per i moduli, presenta un component di root, il cui compito è collegare la gerarchia dei component con il DOM. I component gestiscono la logica applicativa e i dati, rendendo disponibili quest'ultimi al template. Il template rappresenta la vista del component, ovvero come quest'ultimo deve essere renderizzato. Il template è composto da tag HTML e dalla sintassi Angular. Di seguito un'immagine dell'architettura di Angular tratta dal sito ufficiale.

La sintassi Angular definita all'interno delle pagine HTML, permette di modificare gli elementi prima che questi vengano visualizzati. Angular consente di effettuare il binding tra i dati e il DOM. Il binding è un meccanismo per coordinare la comunicazione tra un component e il suo template. I tipi di binding possibili in angular sono tre:

- **Property Binding:** interpola i dati dell'applicazione nel codice HTML. La comunicazione è dal component verso il DOM

- **Event Binding:** a un evento è possibile associarse un handler attraverso il quale aggiornare i dati dell'applicazione. La comunicazione è dal DOM verso il component
- **Two-Way Data Binding:** unione dei precedenti binding. La comunicazione è bidirezionale

Prima che la pagina venga renderizzata, Angular valuta le *direttive* e risolve la propria sintassi modificando il template, sulla base della logica del programma e dei dati. Le direttive possono essere:

- **Components:** direttive che presentano un template
- **Strutturali:** modificano il layout del DOM aggiungendo e/o rimuovendo elementi HTML
- **dell'Attributo:** modificano l'aspetto e/o il comportamento di un elemento, componente o altra direttiva

Angular permette di definire i *Servizi*. Essi consentono di specificare la logica applicativa e i dati senza definire un template associato e, in più, possono essere utilizzati da uno o più components.

Capitolo 4

Il progetto: implementazione del body gateway

Il seguente capitolo tratta i vari aspetti legati all'implementazione e alla realizzazione del body gateway, spiegando da un punto di vista tecnico la realizzazione del sistema e giustificando le varie scelte implementative. L'importanza di tale componente non è da sottovalutare: in quanto consente all'applicazione Android di acquisire automaticamente i dati provenienti dalla body area network, evitando un inserimento manuale dei dati da parte del paziente.

Prima di affrontare gli aspetti implementativi di questa componente, il capitolo offre una panoramica sui device medici attualmente in commercio, i quali permettono di comporre una body area network per pazienti diabetici.
Il capitolo prosegue con un approfondimento sull'hardware selezionato, mentre i restanti paragrafi illustrano i dettagli implementativi di rilievo.

4.1 Device medici presenti sul mercato

Prima di procedere con la progettazione e la realizzazione del body gateway, è stata effettuata un indagine, sui dispositivi smart presenti sul mercato, in grado di aiutare il paziente ad acquisire i dati relativi alla glicemia e alla pressione arteriosa. Poiché il diabete può essere causa di ipertensione, sono state effettuate delle ricerche anche sugli sfigmomanometri.
L'indagine è stata svolta con l'intento di individuare le possibili tecnologie hardware da utilizzare per l'acquisizione dei dati. Si sono presi in conside-

razione i dispositivi in grado di trasmettere i dati a un device esterno e che seguissero uno standard nella trasmissione delle informazioni e/o fornissero delle API per l'accesso ai dati.
Dall'indagine è emerso che, in commercio, sono disponibili diversi dispositivi in grado di trasmettere, via Bluetooth o WiFi, i valori acquisiti a un device esterno. I produttori di tali soluzioni forniscono, inoltre, un'applicazione per device mobili attraverso la quale ricevere i dati. Tuttavia, trattandosi di soluzioni commerciali, i produttori non forniscono alcuna informazione su come la trasmissione avviene e/o non forniscono API per consentire l'accesso ai dati senza l'utilizzo dell'applicazione da loro fornita.
Di seguito sono presenti le tabelle che riassumono le caratteristiche dei device analizzati.

Sfigmomanometri				
Nome	**App nativa**	**API per sviluppo app di terze parti**	**Connettività**	**Accuratezza**
Nokia BPM	IOS, Android	✓	Bluetooth Low Energy	+/- 3 mmHg
iHealth Track	IOS, Android	✓	Bluetooth Low Energy	
iHealth View	IOS, Android	✓	Bluetooth 3.0	+/- 3 mmHg
iHealth Sense	IOS	✓	Bluetooth 3.0	+/- 3 mmHg
iHealth Feel	IOS, Android	✓	Bluetooth 3.0	+/- 3 mmHg
Qardio Arm	IOS, Android		Bluetooth Low Energy	+/- 3 mmHg
Koogeek BP2	IOS, Android		Bluetooth Low Energy, WiFi	
Beurer BM95	IOS, Android		Bluetooth Low Energy	+/- 3 mmHg

Beurer BM77	IOS, Android		Bluetooth Low Energy	+/- 3 mmHg
Beurer BM85	IOS, Android		Bluetooth Low Energy	+/- 3 mmHg
Beurer BM57	IOS, Android		Bluetooth Low Energy	+/- 3 mmHg
Beurer BC85	IOS, Android		Bluetooth Low Energy	+/- 3 mmHg
Beurer BC57	IOS, Android		Bluetooth Low Energy	+/- 3 mmHg
Omorn M7	IOS, Android	✓	Bluetooth Low Energy	+/- 3 mmHg
Omorn MIT5s	IOS, Android	✓	Bluetooth Low Energy	+/- 3 mmHg

Glucometri				
Nome	**App nativa**	**API per sviluppo app di terze parti**	**Connettività**	**Accuratezza**
iHealth Smart	IOS, Android	✓	Bluetooth 3.0	
iHealth Aligh	IOS, Android	✓	Ingresso audio	
OneTouch Verio Flex	IOS, Android		Bluetooth Low Energy	
Contour next one	IOS, Android		Bluetooth Low Energy	+/- 5/15 mg/dL
Gluconext*	IOS, Android		Bluetooth Low Energy, NFC	
* consente di rendere "smart" una serie di glucometri che non sono dotati di tecnologia Bluetooth o altri metodi per la trasmissione dei dati				

Come si evince dalle tabelle, alcuni produttori forniscono delle API per

consentire lo sviluppo di applicazioni di terze parti. Le API consentono di accedere ai dati raccolti dai device, ma per utilizzarli, è necessario inviare delle richieste HTTP al cloud sul quale le informazioni vengono memorizzate. L'invio dei dati al cloud avviene attraverso l'utilizzo dell'applicazione nativa fornita dal produttore. Pertanto, la misurazione della pressione e/o della glicemia deve avvenire tramite l'utilizzo dell'applicazione fornita dai produttori. Poiché il nostro scopo è acquisire i dati direttamente dai device, senza l'utilizzo dell'applicazione fornita dai produttori, tali soluzioni non risultano essere adatte.
Ampliando la ricerca ai sensori biomedici in commercio, da collegare a microcontrollori e/o single board computer, tra le possibili soluzioni è necessario segnalare la piattaforma hardware *e-Health*; la quale consente l'acquisizione di alcuni dati biomedici per la realizzazione di applicazioni mediche e/o biomediche.

4.2 Piattaforma e-Health

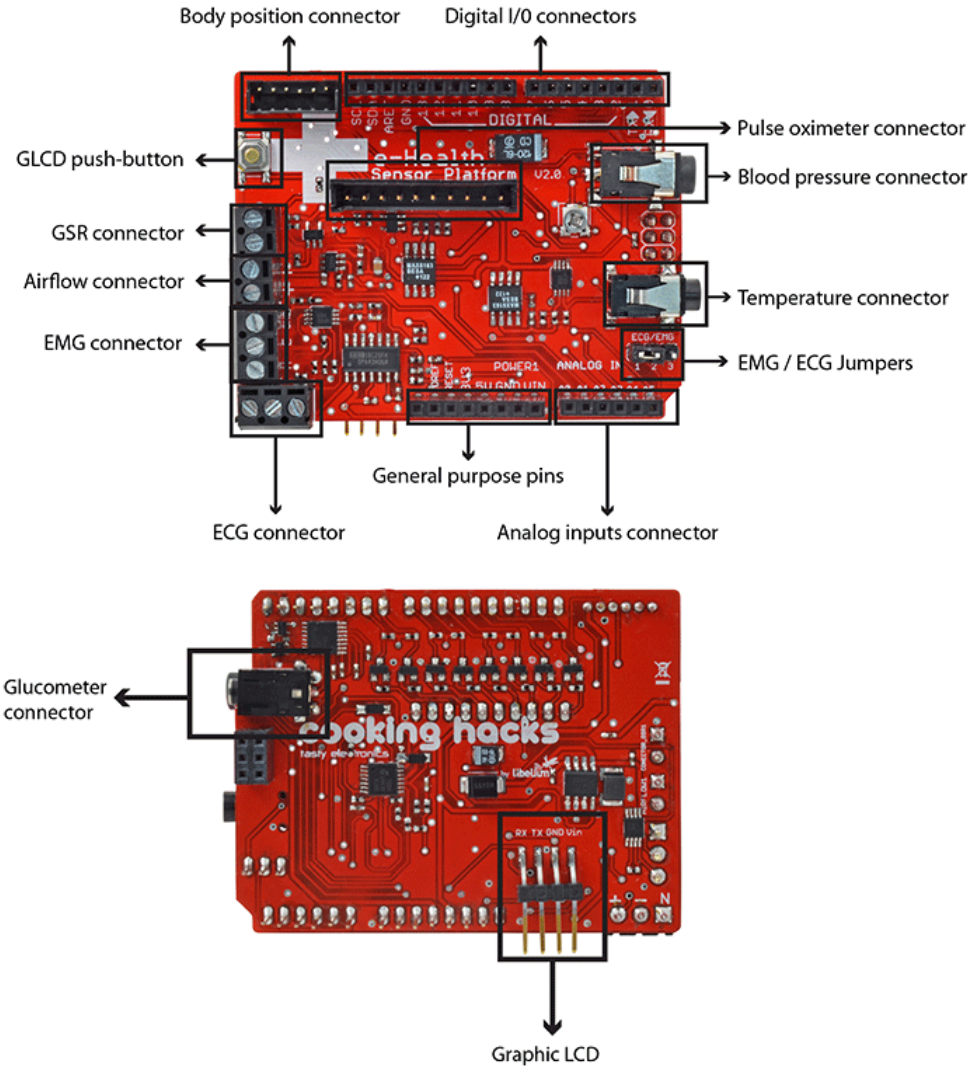

Figura 4.1: Shield e-Health[14]

La piattaforma *e-Health*, realizzata da Cooking-Hacks, è una shield compatibile con Arduino e Raspberry Pi, concepita per l'acquisizione dei dati da alcuni sensori biomedici. Una shield è un circuito stampato in grado di estendere le funzionalità di un microcontrollore o di un single board computer.

La shield presente nella figura precedente è compatibile con la scheda di prototipazione Arduino. Per consentire l'utilizzo della piattaforma su un Raspberry Pi, è necessario utilizzare, abbinata con la piattaforma e-Health, la shield presente nella figura seguente.

Figura 4.2: Shield Connection Bridge[14]

La piattaforma e-Health V. 2.0 consente la trasmissione dei dati attraverso: Wi-Fi, 3G, GPRS, Bluetooth, ZigBee e 802.15.4.

4.2.1 Raspberry Pi

Il Raspberry Pi è un single board computer dalle dimensioni ridotte, realizzato dalla Raspberry Pi Foundation con lo scopo di facilitare l'insegnamento della computer science nelle scuole e nei paesi in via di sviluppo.
Nel corso degli anni sono stati realizzati diversi modelli con caratteristiche hardware differenti.
Le diverse versioni di Raspberry Pi presentano dei pin per fornire un'alimentazione in uscita (3.3V o 5V) e dei GPIO (General Purpose I/O) pin. I GPIO

sono pin digitali e, a seconda della modalità (output o input), è possibile collegare ad essi degli attuatori o dei sensori. Alcuni dei GPIO pin permettono l'utilizzo dei protocolli seriali SPI e I2C.
Tramite un convertitore analogico/digitale è possibile utilizzare dei trasduttori analogici. Inoltre, tramite opportune librerie software, i GPIO pin possono essere utilizzati come pin PWM (Pulse Width Modulation).

4.2.2 Arduino

Arduino è una piattaforma di sviluppo open source basata su un microcontrollore. All'hardware viene affiancato un ambiente di sviluppo integrato multipiattaforma.
I vari modelli Arduino sono dotati di funzionalità di I/O. I connettori sono suddivisi in I/O digitale, di cui una parte possono produrre segnali in modulazione di frequenza (PWM) e altri possono essere utilizzati come input analogici. Gli input analogici possono essere riprogrammati per operare come normali pin di I/O digitali. Per poter leggere i valori analogici, le schede Arduino sono dotate di un convertitore analogico-digitale. La scheda Arduino Due è inoltre dotata di due convertitori digitale-analogico, in modo da avere due pin analogici di output.
Nel corso degli anni sono state immesse sul mercato varie tipologie di schede, con caratteristiche hardware simili, a seconda dello scopo specifico. Sono ora disponibili soluzioni nelle fasce: entry level, Internet of Things, wearable e stampa 3D.

4.2.3 Sensori

I sensori supportati dalla piattaforma consentono di acquisire le seguenti informazioni:

- Frequenza cardiaca
- Percentuale di ossigeno nel sangue
- Temperatura corporea
- Glicemia
- Pressione arteriosa

- Risposta galvanica della pelle
- Respirazione
- Posizione del corpo
- ECG (Elettrocardiogramma)
- EMG (Elettromiografia)

Figura 4.3: Sensori disponibili con la piattaforma[14]

Software

Per semplificare l'acquisizione dei dati dai sensori tramite la piattaforma e-Health, Cooking-Hacks fornisce una libreria realizzata in C++. Viene fornita inoltre la libreria *ArduPi*. Tale libreria consente agli sviluppatori di utilizzare il medesimo codice C++ sia su Raspberry Pi che su Arduino.
All'interno del PSLab, con lo scopo di facilitare la realizzazione di sistemi software in ambito biomedico, è stata realizzata la libreria Java *HealthPi* [15].

4.2.4 Libreria HealthPi

La libreria Java *HealthPi* [15] è stata realizzata con lo scopo di facilitare lo sviluppo di sistemi che acquisiscono dati biomedici tramite la piattaforma e-Health.
La libreria realizzata da [15], per accedere ai dati dei sensori della piattaforma e-Health, utilizza la libreria fornita dal produttore della piattaforma attraverso l'ausilio del framework JNI.
La libreria Java consente di accedere ai dati della piattaforma tramite procedure call. Oltre all'utilizzo di quest'ultime, è possibile definire un observer per la particolare informazione a cui siamo interessati.
Di seguito sono presenti due metodi per l'acquisizione dei dati tramite procedure call e observer.

```
1  IHealthPlatform platform = new HealthPlatform();
2  IGlucoseLevel glucose=platform.getGlucoseConcentration();
3  System.out.println(glucose.getValue());
```

```
1  ObservableSensorsFactory factory=new ObservableSensorsFactory();
2  IObservablePulseSensor
       observable=factory.createObservablePulseSensor(1000);
3
4  observable.register(new IObserver() {
5     @Override
6     public void update(Observable o, Object arg) {
7        System.out.println("Heart rate: "+((HeartRate)arg).getValue()+"
             BPM");
8     }
9  });
10
11 observable.start();
```

Come visibile dal listato, la libreria modella i dati acquisiti dai sensori tramite dei POJO. La libreria, inoltre, al fine di facilitare la trasmissione e il salvataggio dei dati, consente di modellare i dati in formato Protocol Buffers o in formato JSON. Protocol Buffers è un linguaggio naturale, platform independent ed estensibile, realizzato da Google per la serializzazione dei dati [16].
La libreria definisce delle classi che rappresentano i dati biomedici relativi alla: pressione arteriosa, risposta galvanica della pelle, concentrazione di glucosio, temperatura corporea, respirazione, ECG, EMG, percentuale di ossigeno nel sangue, frequenza cardiaca e posizione del corpo.

4.3 L'hardware selezionato: Raspberry Pi e piattaforma e-Health

Il primo step fondamentale per lo sviluppo del body gateway consiste nella selezione dell'hardware. Altro aspetto importante è la scelta dei sensori che costituiranno la body area network.
Sebbene i requisiti funzionali del body gateway non siano particolarmente complessi, è bene valutare attentamente anche i requisiti non funzionali; al fine di poter selezionare la base computazionale che meglio li soddisfa. I requisiti che l'hardware deve soddisfare, sono:

- Il device deve essere economico
- Il device deve essere facilmente manutenibile
- Il device deve avere bassi consumi energetici
- Il device deve essere di dimensioni ridotte; in quanto il paziente dovrà indossarlo
- Il device deve comunicare con il dispositivo mobile del paziente via Bluetooth
- Il device deve essere modulare e facilmente estendibile

Sulla base di tali requisiti, la scelta della piattaforma hardware ricade sul Raspberry Pi. La scelta di utilizzare tale piattaforma è motivata dalla possibilità di sviluppare un prodotto software a un livello più elevato rispetto ad altri microcontrollori e/o single board computer presenti sul mercato, come ad esempio Arduino e PIC. Inoltre, Raspberry Pi fornisce una maggiore potenza di calcolo e la possibilità del multi-thread nativo.
Una volta definito l'hardware atto a ricoprire il ruolo di body gateway, è necessario selezionare i sensori smart che dovranno comporre la body area network. Sulla base di quanto detto nei paragrafi precedenti, la scelta ricade sulla piattaforma e-Health; in quanto è perfettamente integrabile con Raspberry Pi e offre un accesso diretto ai dati dei sensori presenti nel kit.

4.4 Connessione e trasferimento dati

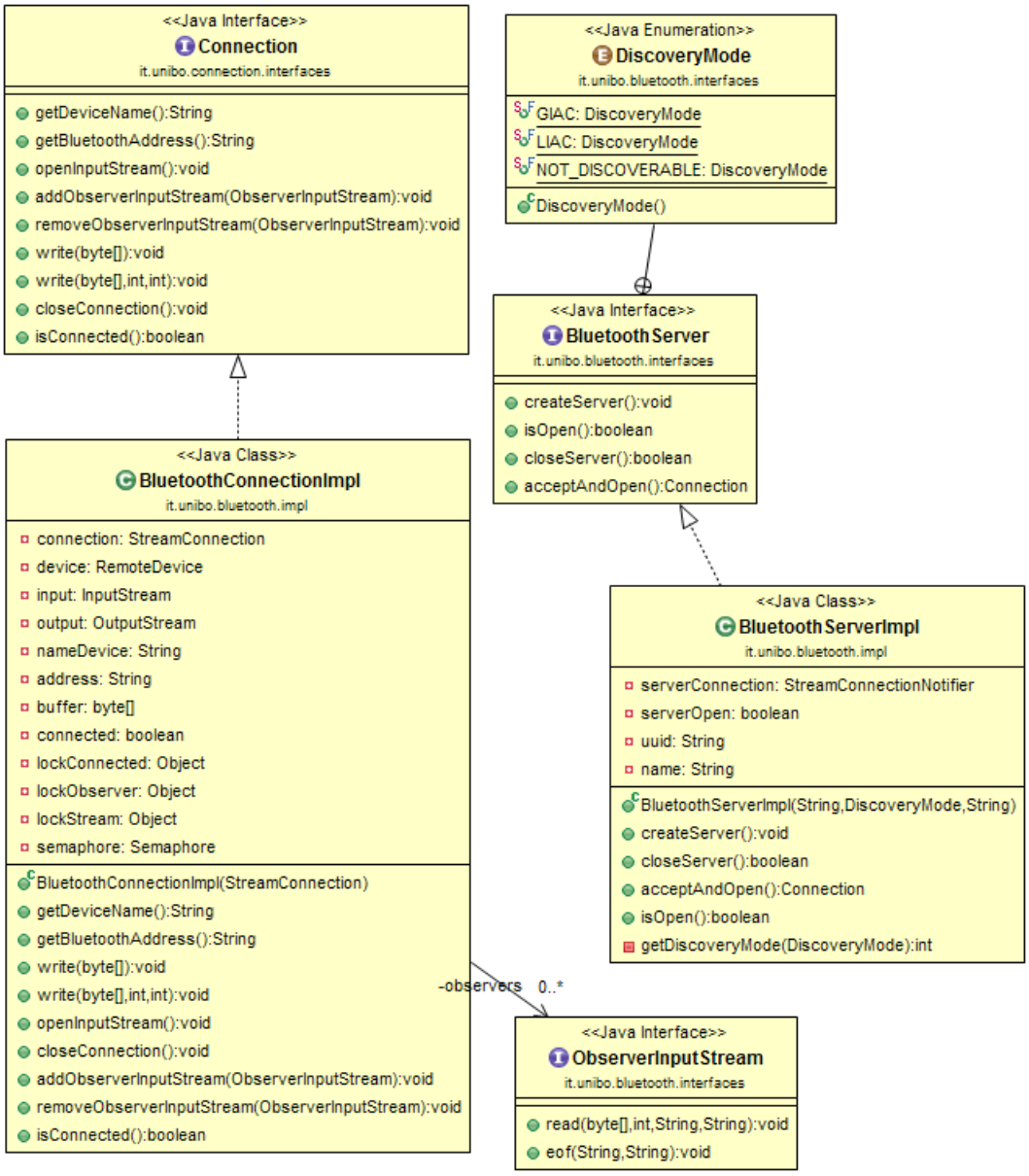

Figura 4.4: Diagramma delle classi per la connessione e trasferimento dei dati via Bluetooth

Sulla base dell'architettura logica, il body gateway ha il compito di accettare la richiesta di connessione Bluetooth effettuata dallo smartphone. Pertanto, esso avrà il ruolo di server e dovrà essere visibile ai device che avranno intenzione di effettuare il pairing.
In base alla tecnologia scelta, non è presente alcuna API nativa che consente di gestire le connessioni Bluetooth. Pertanto, per la gestione della connessione Bluetooth e del trasferimento dei dati attraverso questa tecnologia di comunicazione, è necessario realizzare un'API. Per la realizzazione, si è deciso di avvalersi della libreria *Bluecove*, la quale fornisce gli strumenti necessari per gestire una connessione e trasferire dei dati via Bluetooth.
Sulla base di quanto detto, è possibile definire il diagramma delle classi visibile nella figura 4.4.
L'interfaccia *BluetoothServer* definisce i metodi necessari per permettere al device di agire come server e accettare le connessioni Bluetooth in ingresso.
Come si evince dal nome, la classe *BluetoothServerImpl*, implementa i metodi dell'interfaccia *BluetoothServer* al fine di permettere al body gateway di agire come server e accettare le connessioni Bluetooth in ingresso.
Il costruttore delle classe *BluetoothServerImpl* accetta tre parametri: il nome del server, il livello di visibilità del device e l'uuid.
Il livello di visibilità può essere di due tipi: GIAC e LIAC. Con il livello di visibilità GIAC, il body gateway è visibile a tutti i device Bluetooth. Con il livello di visibilità LIAC, invece, il body gateway è visibile soltanto ai device Bluetooth con cui ha effettuato il pairing in precedenza.
L'UUID è un identificatore univoco che rappresenta il tipo di servizio offerto da un device Bluetooth.
Tramite il metodo *createServer*, si crea il server Bluetooth attraverso il quale è possibile accettare le connessioni in ingresso.
Tramite il metodo *acceptAndOpen*, è possibile porsi in attesa di una connessione Bluetooth. Quando quest'ultima si verifica, il metodo restituisce un oggetto della classe *BluetoothConnectionImpl*. Attraverso questo oggetto, è possibile scambiare dei dati con il device con cui si è stabilita la connessione.

La classe *BluetoothConnectionImpl* implementa l'interfaccia *Connection*, la quale definisce i metodi per permettere la comunicazione tra i due device; attraverso qualsiasi tecnologia di comunicazione. Nel nostro caso, poiché siamo interessati a un trasferimento dati via Bluetooth, si implementata l'interfaccia al fine di permettere una comunicazione tramite questa tecnologia.

Il metodo *write* consente di inviare dei byte al device con il quale si è connessi. L'invio dei dati può fallire solo nel caso in cui lo stream sia chiuso. Tale situazione si verifica solo nel caso in cui la connessione Bluetooth sia terminata. Pertanto, al verificarsi di questa situazione, si procede con la chiusura della connessione; al fine di liberare le risorse.
Tramite il metodo *openInputStream* è possibile aprire lo stream in input al fine di leggere i dati ricevuti. Poiché il metodo *read* della classe *InputStream* è bloccante, per evitare di bloccare il flusso di controllo in attesa di ricevere dei byte, si crea un thread; il cui compito è ricevere i dati fino a quando la connessione non termina. Alla ricezione dei byte o alla chiusura della connessione, si effettua una procedure call sugli observer che è possibile aggiungere tramite il metodo *addObserverInputStream*.
La classe è thread safe, al fine di consentire lo sviluppo concorrente senza che lo sviluppatore definisca i meccanismi di mutua esclusione.
Poiché la classe BluetoothConnectionImpl non interpreta in alcun modo i dati che vengono scambiati tra i device, la libreria può essere utilizzata in molteplici contesti in cui è richiesto lo scambio di dati via Bluetooth.

4.5 Ricezione e invio dati biomedici

Il modulo realizzato, tramite l'ausilio della libreria *Bluecove*, consente al body gateway di comunicare con un device via Bluetooth a basso livello, attraverso lo scambio di byte.
Il body gateway interagisce con il device del paziente attraverso la ricezione di richieste di acquisizione dati, e l'invio di quest'ultimi. La libreria realizzata nella sezione precedente, rende complessa questa operazione, in quanto la comunicazione si basa su dei byte. Per fornire una comunicazione ad alto livello, è necessario realizzare un'API che ne consenta l'interpretazione. Il diagramma delle classi presente nella figura seguente definisce le entità necessarie per consentire lo scambio di dati tra i due device.
La classe *PlatformEHealthDataImpl* fornisce i metodi necessari per ricevere le richieste di acquisizione dati e, l'invio di quest'ultimi, al device con il quale si è connessi.

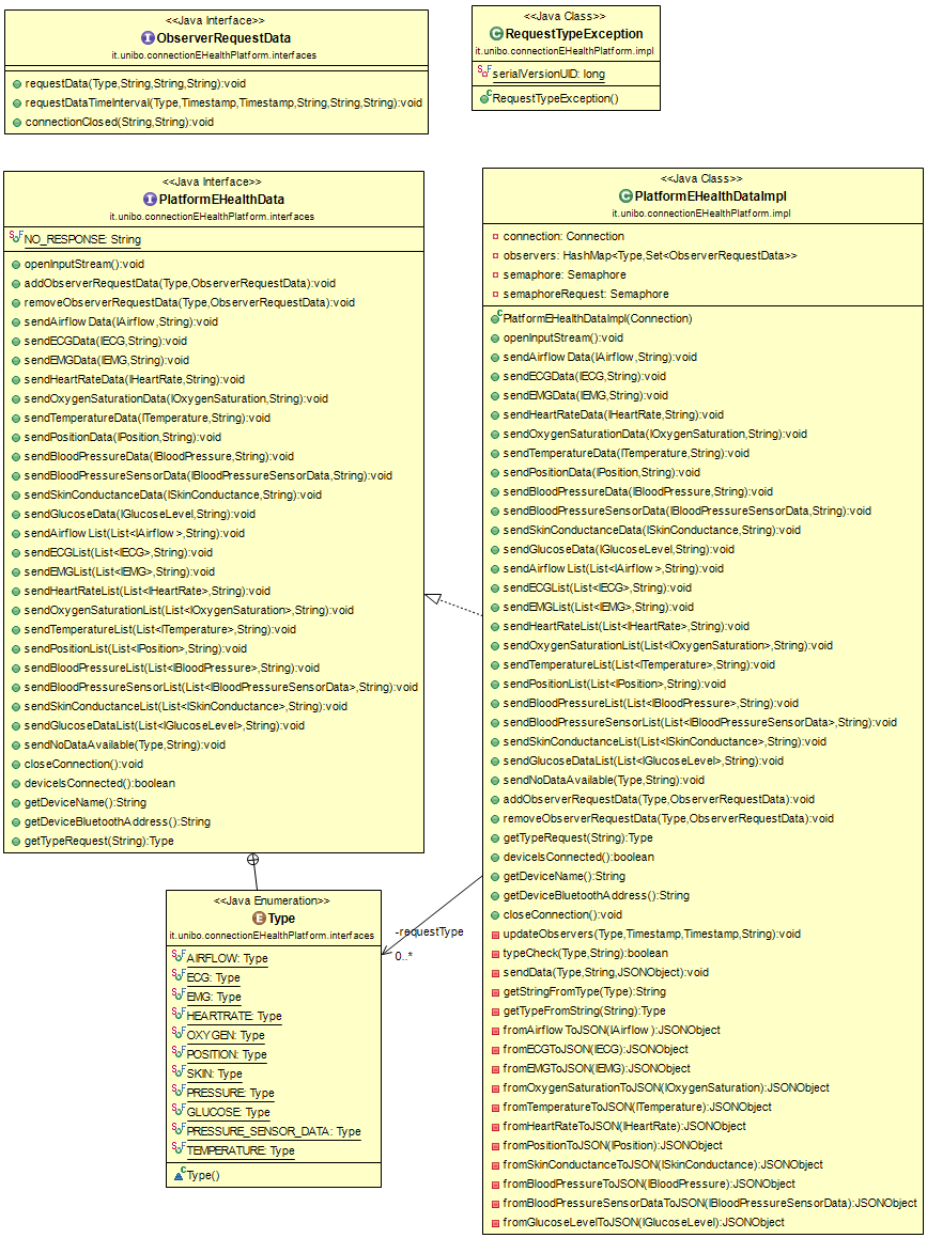

Figura 4.5: Diagramma delle classi per la ricezione e l'invio dei dati

4.5.1 Ricezione richieste acquisizione dati

Il costruttore della classe accetta come parametro l'oggetto che implementa la classe *Connection*; ovvero l'oggetto che rappresenta la connessione con un device. In questo modo, è possibile ricevere le richieste di acquisizione dati e inviare quest'ultimi attraverso diverse tecnologie di comunicazione; non obbligando lo sviluppatore a utilizzare la tecnologia Bluetooth.
Al momento della creazione dell'oggetto, si aggiunge un observer per la lettura dei dati in arrivo. Ogni qualvolta che viene invocato il metodo *read*, i byte vengono interpretati come char. Fino a quando non viene letto il carattere di nuova riga ('\n'), i caratteri letti fino a quel momento vengono concatenati. Al momento della ricezione del carattere di nuova riga, la stringa formatosi fino a quel momento, viene interpretata secondo la sintassi JSON. Se la stringa è compatibile con la sintassi JSON, si tenta di interpretare la richiesta di acquisizione dati. L'oggetto JSON è formato dai campi:

- **type**: indica il tipo di dato richiesto
- **requestCode**: identificatore univoco della richiesta
- **from** e **to**: campi opzionali che indicano l'intervallo di tempo in cui i dati sono stati rilevati

Una volta identificato il tipo di dato richiesto, se presenti, si effettuano le procedure call sui relativi observer. Per ogni tipo di dato è possibile aggiungere un observer, in modo tale che lo sviluppatore possa ricevere solo le richieste a cui è interessato.
Agli observer non viene segnalata solo la ricezione di una richiesta di acquisizione dati, ma anche quando la connessione Bluetooth termina.

4.5.2 Invio dati

I metodi contrassegnati dal prefisso *send*, consentono l'invio dei dati. È possibile inviare un singolo valore o una lista per un particolare tipo di dato. Inoltre, è possibile informare il device con il quale si è connessi, che un tipo di dato non è al momento disponibile, attraverso il metodo *sendNoDataAvailable*.
Come per la ricezione delle richieste, i dati vengono inviati secondo la sintassi JSON, la cui struttura è la seguente:

- **data**: nel caso in cui venga inviato un singolo valore relativo a un dato biomedico, il JSON conterrà questo campo, il quale è a sua volta un JSON che rappresenta il parsing del POJO che modella il dato inviato
- **list**: nel caso in cui venga invita una lista di valori relativi a un dato biomedico, il JSON conterrà questo campo, il quale è rappresentato da un JSONArray in cui ogni elemento è un JSON che rappresenta il parsing del POJO che modella il dato inviato
- **type**: indica il tipo di dato richiesto
- **requestCode**: identificatore univoco della richiesta di acquisizione dati (se presente)

Si tiene a segnalare che tale classe può essere utilizzata indipendentemente dal tipo di device da cui si acquisiscono i dati; in quanto le classi che modellano i dati, sono completamente indipendenti dai possibili device da cui vengono acquisiti.
Nonostante l'architettura logica preveda che il body gateway possa inviare i dati in seguito a una richiesta di acquisizione, l'API realizzata consente l'invio dei dati anche nel caso in cui non si sia verificata alcuna richiesta.

4.6 Il sistema

Una volta realizzate le API, necessarie per semplificare la realizzazione della funzione del body gateway, è possibile implementare la soluzione sulla base dell'architettura logica. Come illustrato nel diagramma di sequenza nel paragrafo relativo all'architettura logica, il body gateway si pone in attesa di ricevere le richieste di pairing via Bluetooth. Stabilita la connessione, il gateway, a ogni richiesta di acquisizione dati, recupera i dati dai sensori e li invia al device con il quale è connesso.

```
1 public class Main {
2    private static BluetoothServer server;
3    private static HealthPlatform platform;
4    private static BluetoothEHealthDataImpl bluetoothHealthData;
5
6    public static void main(String[] args) throws Exception {
7       platform=new HealthPlatform();
8       ObserverRequestData observer=new ObserverRequestData() {
```

```
9           @Override
10          public void requestData(Type type, String requestCode,String
                nameDevice,String address) {
11             if(type==Type.HEARTRATE){
12                bluetoothHealthData.sendHeartRateData(
                      platform.getPulse(),requestCode);
13             }
14          }
15
16          @Override
17          public void requestDataTimeInterval(Type type, Timestamp from,
                Timestamp to, String requestCode,String nameDevice,String
                address) {
18             try {
19                switch(type) {
20                   case PRESSURE_SENSOR_DATA:{
21                      List<IBloodPressureSensorData>
                            data=platform.getBloodPressureSensorDataSequence();
22                      List<IBloodPressureSensorData> values=new ArrayList<>();
23                      for(int i=0;i<data.size();i++) {
24                         BloodPressureSensorData
                               pressure=(BloodPressureSensorData) data.get(i);
25                         Timestamp pressureTimestamp=(Timestamp)
                               pressure.getBloodPressure().getTimestamp();
26                         if(pressureTimestamp.compareTo(from)>=0 &&
                               pressureTimestamp.compareTo(to)<=0) {
27                            values.add(pressure);
28                         }
29                      }
30
31                      if(values.size()>0) {
32                         bluetoothHealthData.sendBloodPressureSensorList(
                               values, requestCode);
33                      }else {
34                         bluetoothHealthData.sendNoDataAvailable(
                               Type.PRESSURE_SENSOR_DATA, requestCode);
35                      }
36                   }break;
37
38                   case GLUCOSE:{
39                      List<IGlucoseLevel>
                            data=platform.getGlucoseSensorDataSequence();
40                      List<IGlucoseLevel> values=new ArrayList<>();
41                      for(int i=0;i<data.size();i++) {
42                         GlucoseLevel glucose=(GlucoseLevel)data.get(i);
43                         Timestamp glucoseTimestamp=(Timestamp)
                               glucose.getTimestamp();
44                         if(glucoseTimestamp.compareTo(from)>=0 &&
                               glucoseTimestamp.compareTo(to)<=0) {
45                            values.add(glucose);
46                         }
47                      }
48
```

```
49                      if(values.size()>0) {
50                         bluetoothHealthData.sendGlucoseDataList(values,
                               requestCode);
51                      }else {
52                         bluetoothHealthData.sendNoDataAvailable(Type.GLUCOSE,
                               requestCode);
53                      }
54                   }break;
55
56               }
57            }catch(Exception e) {
58               e.printStackTrace();
59            }
60
61         }
62
63         @Override
64         public void connectionClosed(String nameDevice, String address) {
65         }
66
67      };
68
69      while(true) {
70         server=new BluetoothServerImpl("04c6093b00001000800000805f9b34fb",
               DiscoveryMode.GIAC,"BluetoothServer");
71         server.createServer();
72         BluetoothConnection connection=server.acceptAndOpen();
73         server.closeServer();
74         bluetoothHealthData=new BluetoothEHealthDataImpl(connection);
75         bluetoothHealthData.addObserverRequestData(
               Type.PRESSURE_SENSOR_DATA, observer);
76         bluetoothHealthData.addObserverRequestData(Type.GLUCOSE, observer);
77         bluetoothHealthData.addObserverRequestData(Type.HEARTRATE,
               observer);
78         connection.openInputStream();
79
80         while(bluetoothHealthData.deviceIsConnected()) {
81            Thread.sleep(10000);
82         }
83      }
84   }
85 }
```

Si procede con la creazione del server Bluetooth e si attende la ricezione di una connessione. Al verificarsi della connessione, si chiude il server e si aggiunge un observer per ricevere le richieste di acquisizione dati relativi a: pressione arteriosa, concentrazione di glucosio nel sangue e frequenza cardiaca. Il server viene chiuso per impedire il pairing con altri device Bluetooth durante la connessione attualmente in corso. Quando la connessione con il device termina, si crea nuovamente il server e ci si pone in attesa di una nuova connessione.

Ogni qualvolta che viene effettuata una callback sull'observer definito, il body gateway ha ricevuto una richiesta di acquisizione dati. Si controlla dunque il tipo di dato richiesto e si recuperano le relative informazioni. Poiché per le misurazioni relative alla pressione arteriosa e alla glicemia, i device sulla quale sono presenti le informazioni sono dotati di memoria, si prendono in considerazione soltanto le richieste in cui è presente l'intervallo di tempo in cui sono state effettuate le misurazioni.

4.7 Deployment

Per eseguire il programma sul Raspberry Pi è necessaria una versione del device precedente alla tre, e la piattaforma e-Health per l'acquisizione dei dati biomedici. Le versioni successive alla due del Raspberry Pi non sono adatte; in quanto il pulsiossimetro, necessario per rilevare la frequenza cardiaca, non supporta le versioni successive.
Poiché le versioni precedenti alla tre del Raspberry Pi sono sprovviste di tecnologia Bluetooth, è necessario dotare il device di un adattatore. In più, è necessario installare *BlueZ*: il supporto ufficiale Linux del protocollo Bluetooth.
Prima di avviare il programma sul Raspberry Pi, è necessario eseguire le seguenti istruzioni:

- Aprire un terminale
- Eseguire il comando: *sudo systemctl start bluetooth* per avviare il servizio Bluetooth
- Eseguire il comando: *sudo bluetoothctl*
- Eseguire il comando: *power on* per attivare il Bluetooth
- Eseguire il comando: *pairable on* per consentire l'accoppiamento Bluetooth tra i device
- Se si vuole evitare di confermare il codice per l'accoppiamento, eseguire il comando: *agent NoInputNoOutput*. Tale comando può essere utile quando non è possibile collegare il Raspberry Pi a un monitor.
- Eseguire il comando: *default-agent*

Se si vuole evitare l'intervento umano per consentire al Raspberry Pi di ricevere connessioni Bluetooth e inviare dati biomedici, può essere utile eseguire le istruzioni elencate in precedenza e lanciare il programma al momento del boot del Raspberry Pi. Per fare ciò è sufficiente:

- Aprire un terminale
- Eseguire il comando: *sudo nano /etc/rc.local*
- Inserire al termine del file, prima dell'istruzione *exit 0* i seguenti comandi:

```
1        sudo systemctl start bluetooth
2        sudo bluetoothctl << EOF
3        power on
4        pairable on
5        agent NoInputNoOutput
6        default-agent
7        EOF
8
9        sudo java -jar <path>/bluetoothServerHealth.jar
```

Capitolo 5

Il progetto: implementazione del server e della web application

In questo capitolo si affronta la progettazione e la realizzazione dell'applicativo attraverso il quale il medico curante può monitorare i propri pazienti in cura. Inoltre, è presente la progettazione e la realizzazione del back-end per la parte server.

Il primo paragrafo si occupa di illustrare la progettazione delle API RESTful.
Il secondo paragrafo illustra la struttura dei documenti che compongono il database NoSQL.
Il terzo paragrafo mostra la generazione delle regole definite dal medico.
Il quarto paragrafo illustra la gestione del routing delle richieste di maggiore rilevanza.
Il capitolo si conclude con l'illustrazione della struttura dell'applicazione web attraverso la quale il medico interagisce con il sistema.

5.1 Progettazione delle API RESTful

Le API che il server, attraverso il web framework Express, deve fornire al client, devono essere semanticamente significative; in quanto ognuna di essa rappresenta l'interfaccia del metodo che il client andrà a sfruttare.

5.1.1 Definizione degli URL

La struttura dell'URL definisce la semantica dell' API RESTful.
Per tutte le API che agiscono sugli utenti, come prima parte dell'URL, si ritiene opportuno utilizzare la preposizione */user/*.
Infine, il metodo HTTP che si intende utilizzare, deve essere contestuale all'operazione che si intende svolgere con la risorsa.
Sulla base di queste regole, si possono definire le seguenti API:

- POST /users/new-user
- POST /users/login
- POST /users/set-token
- GET /user/get-user-list
- GET /user/get-physician-list
- POST /user/send-notification
- GET /data/get-rule-file
- GET /data/get-rule
- POST /data/edit-rule
- POST /data/delete-rule
- POST /data/add-rule
- POST /data/send-data-simulation
- POST /data/send-data
- GET /data/get-info-simulation
- GET /data/get-info-session
- GET /data/get-data-simulation
- GET /data/get-data

- GET /data/:id1_:id2/stream
- GET /data/:id/session
- GET /data/:id/simulation
- POST /data/add-pressure
- POST /data/add-glycemia
- GET /data/get-pressure
- GET /data/get-glycemia

5.1.2 Le query di ricerca previste

All'interno di questo paragrafo sono presenti le query che saranno effettuate sulle varie entità applicative. Nella prima colonna è presente il caso d'uso in cui la query viene effettuata. Nella seconda colonna è descritta la tipologia della query. Infine, nella terza colonna, è presente una descrizione più dettagliata della query.

Caso d'uso	**Query**	**Descrizione**
Registrazione	Conta omonimi	Conta gli utenti omonimi presenti all'interno del database. Tale query è necessaria per calcolare il numero progressivo dello username; in quanto la sua struttura è definita come: nome.cognome#
Registrazione	Nuovo utente	Inserimento di un nuovo paziente o di un nuovo medico all'interno del database
Login	Login	Ricerca l'utente e ne verifica le credenziali di accesso
Aggiornamento token	Set token	Imposta per un dato utente il token che verrà utilizzato per inviare la notifica push tramite Firebase

Lista pazienti	Get user list	Restituisce la lista dei pazienti di un medico
Lista regole paziente	Get rule	Restituisce l'elenco di regole di un dato paziente
Modifica regola	Edit rule	Modifica la regola di un dato paziente
Elimina regola	Delete rule	Elimina una regola di un dato paziente
Aggiungi regola	Add rule	Aggiunge una regola per un dato paziente
Nuova simulazione	Insert simulation info	Inserisce le informazioni primarie di una simulazione effettuata da un paziente
Nuova simulazione	Insert simulation	Inserisce tutti i dati prodotti da una simulazione effettuata da un paziente
Nuova sessione	Insert session info	Inserisce le informazioni primarie di una sessione real time effettuata da un paziente
Nuova sessione	Insert session data	Inserisce i dati di una sessione real time effettuata da un paziente
Nuova sessione	Edit session info	Inserisce la data di fine della sessione real time effettuata da un paziente
Lista simulazioni paziente	Get info simulation	Restituisce l'elenco delle informazioni primarie delle simulazioni effettuate da un particolare paziente
Lista sessioni paziente	Get info session	Restituisce l'elenco delle informazioni primarie delle sessioni real time effettuate da un particolare paziente
Dettaglio simulazione	Get simulation	Restituisce i dettagli di una particolare simulazione effettuata da un paziente

Dettaglio sessione	Get session	Restituisce i dettagli di una particolare sessione real time effettuata da un paziente
Aggiungi pressione	Add pressure	Aggiunge le misurazioni relative alla pressione arteriosa di un dato paziente
Aggiungi glicemia	Add glycemia	Aggiunge le misurazioni relative alla glicemia di un dato paziente
Lista misurazioni pressione paziente	Get pressure	Restituisce l'elenco delle misurazioni relative alla pressione arteriosa di un dato paziente
Lista misurazioni glicemia paziente	Get glycemia	Restituisce l'elenco delle misurazioni relative alla glicemia di un dato paziente

5.2 La struttura dei documenti

Sebbene nei database di tipo NoSQL non sia necessario definire una struttura per i documenti, si è cercato di rispettare comunque una struttura di base. Di seguito, sono presenti i documenti con le relative strutture.

User

Il documento *User* contiene le informazioni relative agli utenti, sia pazienti che medici, presenti nel sistema.

Nome	**Tipo**	**Descrizione**
ID	String	Username dell'utente
Password	String	Password dell'utente
First_name	String	Nome dell'utente
Last_name	String	Cognome dell'utente
Physician	Boolean	Flag che indica se un utente è un medico o un paziente

Token	String	Token del device del paziente a cui inviare la notifica
Physician_username	String	Username del medico curante del paziente
Gender	String	Sesso del paziente
Birth_date	Date	Data di nascita del paziente
CF	String	Codice fiscale del paziente

Simulation info

Tale documento contiene le informazioni principali relative alle simulazioni effettuate dai pazienti.

Nome	**Tipo**	**Descrizione**
Username	String	Username del paziente a cui è associata la simulazione
Date	Date	Data e ora in cui è stata effettuata la simulazione

Session

Tale documento contiene le informazioni principali relative alle sessioni realtime effettuate dai pazienti.

Nome	**Tipo**	**Descrizione**
Username	String	Username del paziente a cui è associata la sessione real time
Start_time	Date	Data e ora in cui è iniziata la sessione real time del paziente
End_time	Date	Data e ora in cui è terminata la sessione real time del paziente

Info

Tale documento contiene i dettagli delle sessioni real time effettuate dei pazienti.

Nome	Tipo	Descrizione
ID_session	String	ID del record relativo alle informazioni primarie della sessione real time
Username	String	Username del paziente a cui è associata la sessione real time
Time	Date	Data e ora in cui è stata effettuato il rilevamento delle informazioni del paziente durante la sessione real time
Glycemia	Number	Valore glicemico del paziente al tempo *Time*
Insulin	Number	Valore insulina del paziente al tempo *Time*
Glucagon	Number	Valore del glucagone del paziente al tempo *Time*
Liver_glycogen	Number	Valore glicogeno fegato del paziente al tempo *Time*
Muscle_glycogen	Number	Valore glicogeno muscolo del paziente al tempo *Time*
Calories	Number	Calorie bruciate dal paziente al tempo *Time*
Meals	JSON	Alimenti ingeriti dal paziente al tempo *Time*

Simulation info

Tale documento contiene i dettagli delle simulazioni effettuate dei pazienti.

Nome	Tipo	Descrizione
ID_session	String	ID del record relativo alle informazioni primarie della sessione real time
Username	String	Username del paziente a cui è associata la simulazione

Date	Date	Data e ora in cui è stata effettuata la simulazione
Step	Number	Step della simulazione in cui sono stati rilevati i dati
Glycemia	Number	Valore glicemico del paziente al tempo *Step*
Insulin	Number	Valore insulina del paziente al tempo *Step*
Glucagon	Number	Valore del glucagone del paziente al tempo *Step*
Calories	Number	Calorie bruciate dal paziente al tempo *Step*
Meals	JSON	Alimenti ingeriti dal paziente al tempo *Step*

Blood pressure

Il documento *Blood pressure* contiene le misurazioni relative alla pressione arteriosa effettuate dai pazienti.

Nome	**Tipo**	**Descrizione**
Username	String	Username del paziente a cui è associata la misurazione relativa alla pressione arteriosa
Time	Date	Data e ora in cui è stata effettuata la misurazione
Sys	Number	Pressione sistolica della misurazione
Dia	Number	Pressione diastolica della misurazione
Bpm	Number	Frequenza cardiaca al momento della misurazione

Glucose

Il documento *Glucose* contiene le misurazioni relative alla glicemia effettuate dai pazienti.

Nome	Tipo	Descrizione
Username	String	Username del paziente a cui è associata la misurazione relativa alla glicemia
Time	Date	Data e ora in cui è stata effettuata la misurazione
Glycemia	Number	Glicemia della misurazione

Rule

Tale documento contiene le regole definite dai medici per i propri pazienti.

Nome	Tipo	Descrizione
Patient	String	Username del paziente a cui è associata la regola
Parameter	JSON	Condizioni in cui applicare la regola
Solution	JSON	Vincoli della soluzione che la regola deve produrre
Name	String	Nome della regola definita dal medico

5.3 Regole

Il sistema di regole ha il compito di analizzare i dati relativi all'andamento glicemico del paziente; al fine di produrre dei suggerimenti per migliorare lo stile di vita di quest'ultimo. Ogni suggerimento è formato da una coppia di valori: il regime alimentare da seguire (basso, medio ed elevato), abbinato con l'intensità dell'attività fisica da svolgere (basso, medio ed elevato).
Il medico curante può definire, per ciascun paziente in cura, delle regole che forniscono un suggerimento riguardo lo stile di vita da seguire. Per ogni regola, può definire l'insieme delle condizioni che devono essere soddisfatte per elaborare i dati del paziente e fornire un suggerimento. È possibile definire più di una condizione. Le condizioni che si possono definire sono:

- Picco glicemico medio post prandiale

- Coefficiente angolare medio tra i picchi glicemici post prandiali e il valore glicemico al momento del pasto
- Coefficiente angolare medio tra i picchi glicemici post prandiali e il minimo valore glicemico registrato tra le due e quattro ore successive al pasto
- Carico glicemico medio della giornata (basso, medio, elevato)
- Intensità attività fisica svolta durante la giornata (basso, medio, elevato)

Qualora una delle condizioni non fosse soddisfatta, non verrà prodotto alcun suggerimento; in quanto i parametri vitali del paziente non soddisfano le condizioni definite dal medico.
Oltre a definire le condizioni in cui la regola deve fornire un suggerimento, il medico può definire anche i vincoli che il suggerimento deve soddisfare. In particolare, può definire:

- Carico glicemico medio giornaliero
- Intensità attività fisica giornaliera

La regola, quando determina i suggerimenti da restituire al paziente, oltre a tenere conto dei vincoli definiti dal medico, considera anche le preferenze espresse dall'assistito. Ad esempio, il paziente può richiedere dei suggerimenti in cui il carico glicemico dei pasti non sia basso.

Le regole vengono definite dal medico attraverso l'applicazione web, mentre il sistema di regole, come già detto all'interno del terzo capitolo, viene definito attraverso Prolog; il quale si occupa di produrre un feedback sulla base dei dati prodotti dall'assistito. Dunque, Il server deve essere in grado di ricevere le richieste del medico e generare, per ogni paziente, un file Prolog al cui interno si definiscono le strutture logiche delle regole.
Le preferenze espresse dall'assistito potrebbero entrare in conflitto con i vincoli definiti dal medico. In tal caso, il sistema non sarebbe in grado di produrre una soluzione ammissibile. Al fine di garantire almeno una soluzione ammissibile, all'interno del file Prolog in cui sono presenti le regole definite dal medico, è necessario inserire una regola che si occupa di controllare la presenza di almeno una soluzione ed, eventualmente, effettuare il rilassamento dei vincoli. Tale regola, denominata *function_call*, deve accettare i seguenti parametri:

- Nome della regola da chiamare
- Lista dei parametri di input. La lista deve essere costituita dai seguenti elementi:
  - Picco glicemico medio post prandiale
  - Coefficiente angolare medio tra i picchi glicemici post prandiali e il valore glicemico al momento del pasto
  - Coefficiente angolare medio tra i picchi glicemici post prandiali e il minimo valore glicemico registrato tra le due e quattro ore successive al pasto
  - Carico glicemico medio della giornata (da 1 a 3)
  - Intensità attività fisica svolta durante la giornata (da 1 a 3)
- Lista dei vincoli espressi dal paziente. La lista deve essere composta da:
  - Carico glicemico minimo dei pasti
  - Carico glicemico massimo dei pasti
  - Intensità attività fisica minima
  - Intensità attività fisica massima
- Criterio con cui effettuare il rilassamento dei vincoli. Per ogni vincolo è necessario definire:
  - Valore di incremento/decremento
  - Valore minimo del vincolo
  - Valore massimo del vincolo
- Variabile da unificare con l'output

La regola si occupa di risolvere la regola definita nel parametro, e di verificare se quest'ultima restituisce o meno un risultato. Qualora non dovesse produrre alcun risultato, la regola *function_call* dovrà effettuare un rilassamento sui vincoli espressi dal paziente; al fine di garantire almeno una soluzione ammissibile.
Di seguito, è presente, sotto forma di pseudocodice, il funzionamento della regola appena descritta.

```
function_call(FunctionName,Parameters,Constraints,
    RelaxationCriteria,Result):-
   call(FunctionName,Parameters,Constraints,X)
   if(X is empty) then
      relax_constraints(Constraints,RelaxationCriteria,NewConstraints)
      function_call(FunctionName,Parameters,NewConstraints,
          RelaxationCriteria,Result)
   else
      Result=X
```

La regola *function_call* si occupa di risolvere le regole definite dal medico, passando i dati relativi allo stato di salute del paziente e i vincoli espressi da esso. Qualora non dovesse essere possibile ottenere una soluzione ammissibile, verrà effettuato un rilassamento su un vincolo espresso dal paziente; il quale verrà incrementato o decrementato in base al valore specificato. Effettuato il rilassamento, si procederà nuovamente con la risoluzione della regola. In caso di esito negativo, si effettuerà nuovamente un rilassamento sull'ultimo vincolo su cui è stato effettuato, fino al raggiungimento del valore minimo/massimo consentito. Una volta raggiunto il valore minimo/massimo consentito, se non è stata ancora trovata una soluzione ammissibile, si effettuerà un rilassamento sul vincolo successivo, per poi procedere con la risoluzione della regola definita dal medico.
Il server, come già ampiamente descritto, si deve occupare di generare il codice Prolog della regola definita dal medico. Nel listato che segue, è definita la struttura della regola Prolog che il server deve generare.

```
checkValue([Glycemia,Alpha,Beta,Meal,Activity], [MinM,MaxM,MinL,MaxL],
    X):-
   ParameterConstraint1>=MinValueConstraint1,
       ParameterConstraint1=<MaxValueConstraint1, ... ,
       ParameterConstraintN>=MinValueConstraintN,
       ParameterConstraintN=<MaxValueConstraintN,
   meal(M,Im),lifestyle(L,Il),
   Im>=MinM,Im=<MaxM,Il>=MinL,Il=<MaxL,
   ((SolutionConstraint1>=MinValueSolConstraint1,
       SolutionConstraint1=<MaxValueSolConstraint1, ...,
       SolutionConstraintN>=MinValueSolConstraintN,
       SolutionConstraintN=<MaxValueSolConstraintN)),
   X=[M,L].
```

In Prolog, quando si tenta di risolvere una regola o un fatto, la query viene effettuata su tutte le regole o fatti aventi lo stesso nome. Pertanto, tutte le regole definite dal medico, devono avere lo stesso nome. Nel nostro caso, le re-

gole definite dal medico, vengono denominate *checkValue*. Il nome della regola che è possibile definire nell'interfaccia grafica è puramente indicativo.
I *ParameterConstraint* rappresentano i parametri selezionati dal medico attraverso l'interfaccia grafica, mentre *MinValueConstraint* e *MaxValueConstraint* costituiscono i valori che i parametri devono soddisfare affinché la regola possa fornire un suggerimento al paziente. Analogamente, *SolutionConstraint* è definito dal medico e rappresenta il vincolo che la soluzione deve soddisfare.
I parametri *MinM, MaxM, MinL e MaxL* rappresentano i vincoli della soluzione espressi dal paziente.

5.3.1 Implementazione

Dopo aver illustrato il funzionamento generale del sistema di regole, in questa sezione è presente l'implementazione di questa funzionalità. Per ogni paziente, se il medico definisce delle regole, viene associato un file Prolog contenente tutte le regole basate sulla sintassi definita poc'anzi.
In precedenza è stato descritto l'algoritmo con cui deve essere effettuato il rilassamento dei vincoli. Nel listato che segue, è presente l'implementazione.

```
1  function_call(FunctionName,Parameters,Constraints,Relaxation,Result):-
2    call(FunctionName,Parameters,Constraints,List),
3    ((((List\=[])
4    ->Result=List
5    ;(relaxation(Constraints,Relaxation,NewConstraints),
6      NewConstraints\=Constraints,
7      (
8       function_call(FunctionName,Parameters,NewConstraints,Relaxation,Result)
9     )
10   )
11   ),!);Result=[]).
```

La regola *call* è una primitiva di Prolog che consente di risolvere una regola. Nel nostro caso, si risolve la regola *checkValue*; a cui vengono passati le variabili *Parameters*, *Constraints* e *List*
Se la variabile *List* unifica con una lista che non è vuota, allora è possibile risolvere le regole definite dal medico e, pertanto, esiste almeno una soluzione ammissibile. Si procede dunque con unificare la variabile *Result* con *List*. Altrimenti, se *List* è una lista vuota, non esiste alcuna soluzione ammissibile. Dunque, risulta essere necessario un rilassamento dei vincoli.
Tramite la regola *relaxation*, si effettua il rilassamento dei vincoli. Tale regola accetta come termini: la lista dei vincoli, la lista dei criteri con cui ogni vincolo

deve essere rilassato e una variabile con cui unificare la nuova lista di vincoli. Il risultato viene unificato con la variabile *NewConstraints*. Se la lista dei nuovi vincoli coincide con quella di ingresso, allora non è possibile effettuare alcun rilassamento dei vincoli e, pertanto, si restituisce viene restituita una lista vuota. Nel caso in cui dovessero essere diverse, si richiama la regola *function_call* con i vincoli rilassati.
Di seguito è presente l'implementazione della regola *relaxation*.

```
relaxation([Value|[]],[(DescAsc,Min,Max)|[]],Out):-
  NewValue is Value + DescAsc,
  (
  (NewValue >= Min, NewValue =< Max)
  -> Out = [NewValue]
  ; Out = [Value]
  ).

relaxation([Value|Tail],[(DescAsc,Min,Max)|T],Out):-
  NewValue is Value + DescAsc,
  (
  (NewValue >= Min, NewValue =< Max)
  -> Out = [NewValue|Tail]
  ; relaxation(Tail,T,X), Out = [Value|X]
  )
```

Partendo dal caso base, in cui la lista è composta da un solo elemento, si calcola il nuovo valore; sommando o sottraendo a quello attuale il valore di incremento o decremento. Se il nuovo valore del vincolo rientra nel range definito dal criterio con cui effettuare il rilassamento, si restituisce quest'ultimo, altrimenti il valore della variabile *Value*.
Nel caso in cui la lista sia composta da una testa e da una coda, si applica il rilassamento al valore presente in testa alla lista. Se il nuovo valore del vincolo rientra nel range, si restituisce la lista di vincoli con il nuovo valore e, in coda, i restanti vincoli su cui non si effettua alcun rilassamento. Nel caso in cui non dovesse essere possibile rilassare il vincolo, si richiama la regola *relaxation*; passando le code delle liste e la variabile X con cui unificare il risultato. Il risultato restituito dalla regola forma la nuova lista; in cui la testa è formata dal vincolo che non è stato possibile rilassare, e la coda è formata dalla lista presente nella variabile X. Nel caso in cui nessun vincolo possa essere rilassato, la lista di vincoli prodotta sarà uguale a quella di input.
La regola *function_call* consente di elaborare un feedback alla volta. Qualora si vogliano ottenere più feedback, effettuando una sola query, è necessario aggiungere la regola sottostante.

```
get_feedback(_,[],_,_,Result):-Result=[].

get_feedback(FunctionName,[Parameters|T],[Constraints|T2],Relaxation,
    Result):-
function_call(FunctionName,Parameters,Constraints,Relaxation,Tmp),
get_feedback(FunctionName,T,T2,Relaxation,Tmp2),
Result=[Tmp|Tmp2].
```

La regola accetta i medesimi parametri di *function_call*, ma con la differenza che i parametri e i vincoli sono delle liste. Pertanto, si risolve la regola *function_call* per un numero di volte pari alla lunghezza della lista dei parametri. I suggerimenti prodotti per ogni singola risoluzione della regola *function_call* vengono unificati, sotto forma di lista, con la variabile *Result*.
Come già detto, per ogni paziente è necessario creare un file Prolog con le regole definite dal medico curante. Ognuno di essi deve contenere le regole che consentono di risolvere le regole definite dal medico e, all'occorrenza, effettuare il rilassamento dei vincoli.
Al fine di semplificare il processo di creazione, da parte del server, del file Prolog contenente le regole definite dal medico, si vuole realizzare un programma Prolog in grado di creare il file contenente: le regole definite dal medico, le regole necessarie per ottenere il feedback e le regole per effettuare il rilassamento dei vincoli. Di seguito è presente l'implementazione.

```
createPrologFile(Name):-
  open(Name,write,Out),
  write(Out,'get_feedback(_,[],_,_,Result):-Result=[].\n\n'),
  write(Out,'get_feedback(FunctionName,[Parameters|T],[Constraints|T2],
      Relaxation,Result):-\nfunction_call(FunctionName,Parameters,
      Constraints,Relaxation,Tmp),\nget_feedback(FunctionName,T,T2,
      Relaxation,Tmp2),\nResult=[Tmp|Tmp2].\n\n'),
  write(Out,'function_call(FunctionName,Parameters,Constraints,Relaxation,
      Result):-\nfindall(X,call(FunctionName,Parameters,Constraints,X),
      List),!,\n((((List\\=[])\n->Result=List\n;(relaxation(Constraints,
      Relaxation,NewConstraints),\nNewConstraints\=Constraints,\n
      (function_call(FunctionName,Parameters,NewConstraints,
      Relaxation,Result))\n)\n),!);Result=[]).\n\n'),
  write(Out,'checkRelaxation([H|[]],[(Val,Min,Max)|[]],Result):-\n(((
      H==Min,Val<0);(H==Max,Val>0))\n-> Result=false\n;
      Result=true\n).\n\n'),
  write(Out,'checkRelaxation([H|T],[(Val,Min,Max)|T2],Result):-\n
      checkRelaxation(T,T2,X),\n(((H==Min,Val<0);(H==Max,Val>0))\n->
      Tmp=false\n; Tmp=true\n),\n((Tmp==true;X==true)\n-> Result=true\n;
      Result=false\n).\n\n'),
  write(Out,'relaxation([Value|[]],[(DescAsc,Min,Max)|[]],Out):-\n NewValue
      is Value + DescAsc,\n (\n (NewValue >= Min, NewValue =< Max)\n -> Out
```

```
      = [NewValue]\n ; Out = [Value]\n ).\n\n'),
  write(Out,'relaxation([Value|Tail],[(DescAsc,Min,Max)|T],Out):-\n
      NewValue is Value + DescAsc,\n (\n (NewValue >= Min, NewValue =<
      Max)\n -> Out = [NewValue|Tail]\n ; relaxation(Tail,T,X), Out =
      [Value|X]\n ).\n\n'),
  close(Out).


writefact(Name,Fact):-
  open(Name,append,Out),
  write(Out,Fact),
  write(Out,'\n'),
  close(Out).

addRule(NameFile,Name,Parameters,Constraints,Result,Body):-
  open(NameFile,append,Out),
  write(Out,Name),
  write(Out,'(['),
  addParameters(Out,Parameters),
  write(Out,'],['),
  addParameters(Out,Constraints),
  write(Out,'],'),
  write(Out,Result),
  write(Out,'):-\n'),
  write(Out,Body),
  close(Out).

addParameters(_,T):-
  T = [].

addParameters(Out,[H|T]):-
  T = [],
  write(Out,H),!.

addParameters(Out,[H|T]):-
  write(Out,H),
  write(Out,','),
  addParameters(Out,T).
```

Attraverso la regola *createPrologFile* è possibile creare un file Prolog con il nome specificato nella prima variabile. Al momento della creazione, inoltre, si aggiungono l'insieme di regole necessarie per la risoluzione delle regole definite dal medico.
La regola *writeFact* consente di aggiungere un fatto all'interno del file Prolog definito attraverso la variabile *Name*.
La regola *addRule* consente di aggiungere una regola sulla base della struttura definita in precedenza. Di seguito è presente l'elenco dei termini e il loro scopo:

- La prima variabile consiste nel nome del file su cui aggiungere la regola

- La seconda variabile rappresenta il nome della regola
- La terza variabile rappresenta la lista dei parametri che determinano la condizione in cui la regola deve essere applicata
- La quarta variabile rappresenta i vincoli della soluzione della regola
- La quinta variabile rappresenta la variabile su con cui unificare il suggerimento prodotto dalla regola
- La sesta variabile rappresenta il corpo della regola

Invocando questa regola, si aggiungono le regole definite dal medico secondo la struttura illustrata in precedenza. Si tiene a precisare che, la porzione che viene prodotta in automatico, è relativa alla creazione della testa della regola, e non del corpo; il quale deve essere definito nella variabile *Body*

5.3.2 Esempi

Produzione feedback

Assumiamo che il medico, per un dato paziente, abbia definito le seguenti regole:

- Se il picco glicemico medio post-prandiale risulta essere superiore ai 150 mg/dl, il carico glicemico medio dei pasti risulta essere elevato e l'intensità dell'attività fisica svolta risulta essere bassa, i suggerimenti dovranno prevedere un'alimentazione che non sia ad elevato carico glicemico, e un'intensità dell'attività fisica che non sia bassa
- Se il picco glicemico medio post-prandiale risulta essere inferiore ai 90 mg/dl, il carico glicemico medio dei pasti risulta essere basso e l'intensità dell'attività fisica svolta risulta essere elevata, i suggerimenti dovranno prevedere un'alimentazione che non sia a basso carico glicemico, e un'intensità dell'attività fisica bassa

Per analizzare i possibili output del sistema di regole, consideriamo i seguenti casi:

- Paziente con glicemia pari a 185 mg/dl, con un'attività fisica bassa e un'alimentazione ad alto carico glicemico. Il paziente non esprime alcuna preferenza per l'alimentazione e l'attività fisica.

- Paziente con glicemia pari a 80 mg/dl, con un'attività fisica elevata e un'alimentazione a basso carico glicemico. Il paziente esprime come preferenza un'alimentazione a basso carico glicemico. Inoltre, è disposto a svolgere un'attività fisica la cui intensità non è bassa.

- Paziente con glicemia pari a 110 mg/dl, con un'attività fisica bassa e un'alimentazione ad alto carico glicemico. Il paziente non esprime alcuna preferenza per l'alimentazione e l'attività fisica.

Per il primo caso, l'input del sistema di regole è il seguente:

get_feedback(checkValue,[[185,0.4,0.1,3,1]],[[1,3,1,3]],[(-1,1,3),(1,1,3),(-1,1,3),(1,1,3)],X).

Il secondo termine è una lista in cui ogni elemento è costituito dai parametri richiesti dalla regola *checkValue*, mentre il terzo termine è una lista in cui ogni elemento rappresenta i vincoli espressi dal paziente. Il quarto termine rappresenta il criterio con cui rilassare i vincoli.
Sulla base del primo caso, l'output prodotto è il seguente:

[[[low,medium],[low,high],[medium,medium],[medium,high]]]

Al paziente viene suggerito di seguire uno dei seguenti stili di vita:

- Alimentazione a basso carico glicemico e intensità dell'attività fisica media

- Alimentazione a basso carico glicemico e intensità dell'attività fisica elevata

- Alimentazione a medio carico glicemico e intensità dell'attività fisica media

- Alimentazione a medio carico glicemico e intensità dell'attività fisica elevata

Per il secondo caso, l'input del sistema di regole è il seguente:

get_feedback(checkValue,[[80,0.3,0.2,1,3]],[[1,1,2,3]],[(-1,1,3),(1,1,3),(-1,1,3),(1,1,3)],X).

L'output prodotto è il seguente:

[[[medium,low],[high,low]]

Al paziente viene suggerito di seguire uno dei seguenti stili di vita:

- Alimentazione a medio carico glicemico e intensità dell'attività fisica bassa
- Alimentazione a elevato carico glicemico e intensità dell'attività fisica bassa

Come visibile dal risultato, i suggerimenti prodotti dal sistema di regole violano i vincoli espressi dal paziente. Ciò è causato dal rilassamento dei vincoli effettuato dal sistema; in quanto non è stato in grado di trovare una soluzione ammissibile in base ai criteri espressi dall'assistito. Il rilassamento dei vincoli è stato effettuato nel seguente modo:

- Si è tentato di diminuire di uno il livello minimo del carico glicemico dei pasti. Poichè il livello minimo espresso dal paziente era equivalente al valore minimo consentito, si è proseguito con l'aumentare di uno il carico glicemico massimo. Una volta modificato il vincolo, è stata effettuata la risoluzione della regola *checkValue*, passando come vincoli *[1,2,2,3]*
- Il sistema non ha prodotto alcuna soluzione ammissibile. Pertanto, si è proseguito con l'incremento del vincolo relativo al carico glicemico massimo, tentando di risolvere nuovamente la regola *checkValue*, passando come vincoli *[1,3,2,3]*

- Il sistema non ha prodotto alcuna soluzione ammissibile. Pertanto, si è proseguito a diminuire di uno l'attività fisica minima, tentando di risolvere nuovamente la regola *checkValue*, passando come vincoli *[1,3,1,3]*
- Il sistema ha trovato due soluzioni ammissibili

Per l'ultimo caso, l'input del sistema di regole è il seguente:

get_feedback(checkValue,[[110,0.2,0.2,3,1]],[[1,3,1,3]],[(-1,1,3),(1,1,3),(-1,1,3),(1,1,3)],X).

Il sistema non produce alcun suggerimento, in quanto non è stata definita alcuna regola compatibile con i parametri forniti in input.
Come si può notare dagli esempi, il sistema accetta in ingresso delle liste. Pertanto, è possibile elaborare più di un suggerimento. Ad esempio, nel caso in cui il paziente effettui la simulazione della glicemia nell'arco di una settimana, è possibile passare al sistema di regole sette elementi; ogni elemento, sia in input che in output, rappresenta un giorno della settimana. In generale, possiamo dire che l'input del sistema di regole ha la seguente struttura:

get_feedback(checkValue,[[Glycemia1,Alpha1,Beta1,Meal1,Activity1], ...,
[GlycemiaN,AlphaN,BetaN,MealN,ActivityN],
[[MinMeal1,MaxMeal1,MinActivity1,MaxActivity1], ...
[MinMealN,MaxMealN,MinActivityN,MaxActivityN]],
[(-1,1,3),(1,1,3),(-1,1,3),(1,1,3)], X])

L'output prodotto ha la seguente struttura:

[[[MealDay1,ActivityDay1], [MealDay1,ActivityDay1], ...], ..,
[[MealDayN,ActivityDayN], [MealDayN,ActivityDayN], ...]]

Generazione regole

Assumiamo che vogliamo realizzare un programma Prolog per un determinato paziente, attraverso il programma per la generazione del file contenente le regole definite dal medico. Per creare il file, è sufficiente effettuare la seguente query:

createPrologFile('test.pl').

L'esecuzione della query genera un file denominato *test.pl* il cui contenuto è definito nel listato sottostante.

```
get_feedback(_,[],_,_,Result):-Result=[].

get_feedback(FunctionName,[Parameters|T],[Constraints|T2],Relaxation,
    Result):-
function_call(FunctionName,Parameters,Constraints,Relaxation,Tmp),
get_feedback(FunctionName,T,T2,Relaxation,Tmp2),
Result=[Tmp|Tmp2].

function_call(FunctionName,Parameters,Constraints,Relaxation,Result):-
call(FunctionName,Parameters,Constraints,List),
((((List\=[])
->Result=List
;(relaxation(Constraints,Relaxation,NewConstraints),
NewConstraints\=Constraints,
(function_call(FunctionName,Parameters,NewConstraints,Relaxation,Result))
)
),!);Result=[]).

relaxation([Value|[]],[(DescAsc,Min,Max)|[]],Out):-
  NewValue is Value + DescAsc,
  (
  (NewValue >= Min, NewValue =< Max)
  -> Out = [NewValue]
  ; Out = [Value]
  ).

relaxation([Value|Tail],[(DescAsc,Min,Max)|T],Out):-
  NewValue is Value + DescAsc,
  (
  (NewValue >= Min, NewValue =< Max)
  -> Out = [NewValue|Tail]
  ; relaxation(Tail,T,X), Out = [Value|X]
  )
```

Una volta creato il programma Prolog con la porzione necessaria per risolvere le regole definite dal medico e con le regole per effettuare il rilassamento dei vincoli, si vogliono aggiungere dei fatti; necessari per mappare il valore dell'attività fisica e del regime alimentare.

```
writeFact('test.pl','meal(low,1).').
writeFact('test.pl','meal(medium,2).').
writeFact('test.pl','meal(high,3).').
writeFact('test.pl','lifestyle(low,1).').
writeFact('test.pl','lifestyle(medium,2).').
```

```
writeFact('test.pl','lifestyle(high,3).').
```

Attraverso la regola *writeFact*, si aggiungono i fatti *meal* e *lifestyle*.
Infine, si vuole aggiungere la prima regola definita dal medico, dell'esempio precedente, al file *test.pl*. Pertanto bisognerà è necessario effettuare la seguente query:

```
addRule('test.pl','checkValue',['Glycemia','Alpha','Beta','Meal',
    'Activity'],['MinM','MaxM','MinL','MaxL'],'X','Glycemia>=170,
    Glycemia=<300,Meal>=3,Meal=<3,Activity>=1,
    Activity=<1,meal(M,Im),lifestyle(L,Il),Im>=MinM,Im=<MaxM,Il>=MinL,
    Il=<MaxL,Im>=1,Im=<2,Il>=2,Il=<3
   ,X=[M,L].')
```

In questo modo, si genera, all'interno del file *test.pl*, la seguente regola:

```
checkValue([Glycemia,Alpha,Beta,Meal,Activity],[MinM,MaxM,MinL,MaxL],X):-
   Glycemia>=170,Glycemia=<300,Meal>=3,Meal=<3,Activity>=1,Activity=<1,
   meal(M,Im),lifestyle(L,Il),
   Im>=MinM,Im=<MaxM,Il>=MinL,Il=<MaxL,
   ((Im>=1,Im=<2,Il>=2,Il=<3)),
   X=[M,L].
```

5.4 Gestione routing

Attraverso l'ausilio del web framework *Express*, la gestione del routing è estremamente semplice e intuitiva. Di seguito sono presenti le istruzioni necessarie per rendere operativo il server.

```
  var express = require('express')
  var app = express()

  app.use(require('./routes/users'))
  app.use(require('./routes/data'))
```

La keyword *require* è nativa di NodeJs e rappresenta il modo con cui si caricano i moduli. La variabile *app* è un'istanza di *Express*.
Le istruzioni successive permettono di definire i file Javascript in cui sono specificate le route per le richieste HTTP.
La definizione della route ha la seguente struttura:

```
app.METHOD(PATH, HANDLER)
```

Come già detto, *app* è un'istanza di Express, *METHOD* rappresenta il metodo della richiesta HTTP, *PATH* è un percorso sul server e, infine, *HANDLER* è la funzione che viene eseguita quando si trova una corrispondenza per la route. Di seguito sono presenti alcuni esempi su come definire delle route:

```
app.get('/', (req, res)=>{
  res.send('Hello World!');
});

app.post('/post', (req, res)=>{
  res.send('Got a POST request');
});
```

Quando il server riceve una GET sulla home page (url: http://localhost/), invia come risposta un *Hello World!*. Quando riceve una richiesta POST al percorso: *http://localhost/post*, invia come risposta *Got a POST request*.
Attraverso la variabile *req*, è possibile accedere ai dati inviati nelle richieste. Tramite *req.query* è possibile accedere ai parametri della query string e, attraverso *req.body*, è possibile accedere ai parametri presenti nel body della richiesta HTTP.
Come visibile dal listato, definire e gestire le richieste non è particolarmente complesso e, poiché nel nostro caso, la gestione delle richieste definite in precedenza, prevedono delle operazioni su un database e la restituzione di una risposta alle richieste, ci soffermeremo solo su quelle che presentano particolari aspetti implementativi.
Sebbene la gestione delle route sia separata dalla logica di controllo e dal modello dei dati, ai fini della presentazione, verranno mostrati nel medesimo listato.

5.4.1 Registrazione

Come già presentato nel primo paragrafo, la registrazione di un paziente o di un medico, consiste in una richiesta HTTP POST. Di seguito è presente l'implementazione per la gestione delle richieste HTTP POST all'uri */users/new-user*.

```
app.post('/user/new-user',(req, res)=> {
  controller.new_user(req.body.first_name,req.body.last_name,
      req.body.password,req.body.physician,req.body.token,req.
      body.physician_username,req.body.gender,req.body.
      birth_date,req.body.cf, (rescode,data)=>{
```

```
3           res.status(rescode).send(data);
4     });
5   });
6
7   var controller={
8     new_user:(firstname,lastname,password,isPhysician,token,
        physicianUsername,gender,birthdate,cf,result)=>{
9       var username='';
10      model.counts_homonyms(firstname,lastname,(count)=>{
11        if(count===0){
12          username=firstname+'.'+lastname;
13        }else{
14          username=firstname+'.'+lastname+count;
15        }
16
17        model.new_user(username,firstname,lastname,password,
            isPhysician,token,physicianUsername,gender,birthdate
            ,cf,username_callback=>{
18        if(username_callback!==undefined){
19          result(201,{username:username_callback});
20        }else{
21          result(400,{});
22        }
23      })
24    })
25  }
26  }
27
28  var model={
29    counts_homonyms: (firstname,lastname,callback)=>{
30      rethink.db(db_name).table('user').filter({first_name:
          firstname,last_name:lastname}).count().run(connection,(
          err,count)=>{
31        if(err) throw err;
32
33        callback(count);
34      })
35    },
36
37    new_user: (username,firstname,lastname,password,isPhysician,
        token,physicianUsername,gender,birthdate,cf,callback)=>{
38      var user;
39
40      if (isPhysician) {
41        user={
```

```
42          id: username,
43          password,
44          first_name: firstname,
45          last_name: lastname,
46          physician: true
47        };
48      }else{
49        user={
50          id: username,
51          password,
52          first_name: firstname,
53          last_name: lastname,
54          token,
55          physician_username:physicianUsername,
56          physician: false,
57          gender,
58          birth_date:birthdate,
59          cf
60        };
61      }
62
63      rethink.db(db_name).table('user').insert([user]).run(
            connection, err2 => {
64        if(err2) {
65          callback(undefined);
66          return;
67        }
68
69        callback(username);
70      });
71    }
72  }
```

Come visibile dal listato, quando il server riceve una richiesta HTTP POST sulla route definita, effettua una query sul database; al fine di verificare la presenza di omonimi sulla base del nome e del cognome presenti nella richiesta. Come detto poc'anzi, per accedere ai parametri presenti nel body della richiesta, è sufficiente accedere alla proprietà *req.body.[nome_parametro]*.
Se non sono presenti omonimi all'interno del database, il server genera lo username secondo la seguente struttura: *nome.cognome*. In caso contrario, al termine del cognome, si aggiunge un numero che indica il numero di omonimi presenti per quel particolare nome e cognome.
Una volta determinato lo username del nuovo utente, si verifica se quest'ulti-

mo è un paziente o un medico e, in base a questa informazione, si popolano i campi da inserire nel documento. Una volta definiti i campi da inserire, si prosegue con una query di inserimento nel documento *user*. In caso di esito positivo dell'operazione sul database, si invia una risposta alla richiesta HTTP con status code 201 e un JSON contenente lo username.

5.4.2 Lista pazienti

Quando un medico ha intenzione di visualizzare la lista dei pazienti, viene inviata una richiesta HTTP GET.

```
app.get('/user/get-user-list',function(req,res){
    controller.get_user_list(req.query.physician,(rescode,data)
        =>{
        res.status(rescode).send(data);
    });
});

var controller={
  get_user_list: (username,result)=>{
    model.get_user_list(username,(res)=>{
      if(res===false){
        result(400,{});
      }else{
        result(201,res);
      }
    })
  }
}

var model={
  get_user_list: (username,callback)=>{
    rethink.db(db_name).table('user').filter({
        physician_username:username}).orderBy('id').without('
        password').run(connection,(err,list)=>{
      if(err){
        callback(false);
        return;
      }

      callback(list);
    });
```

```
29    }
30 }
```

Ogni qualvolta che il server riceve la richiesta, effettua una query sul documento *user*, filtrando tutti i pazienti che hanno come medico curante quello presente nella query string. Come detto poc'anzi, per accedere ai parametri dell'url, si accede tramite *req.query*. Inoltre, nella query, si chiede di escludere dai campi restituiti la password dei pazienti. In caso di esito positivo della query, si restituisce una risposta con status code 201 e la lista in formato JSON dei pazienti.

5.4.3 Invio notifiche

Per l'invio delle notifiche push, si è scelto di utilizzare Firebase. Su NodeJs è presente il modulo *fcm-node* che consente l'invio delle notifiche tramite Firebase. Per semplificare l'invio delle notifiche, si crea un modulo che utilizza *fcm-node*. Di seguito è presente l'implementazione.

```
1  var FCM = require('fcm-node');
2  var fcm;
3
4  module.exports= {
5      init: function(key){
6          fcm=new FCM(key);
7      },
8
9      sendNotification: function (token, titleNotification,
           bodyNotification, dataNotification) {
10         var message = {
11             to: token,
12             notification: {
13                 title: titleNotification,
14                 body: bodyNotification,
15                 click_action: 'FCM_PLUGIN_ACTIVITY',
16                 icon: "fcm_push_icon"
17             },
18
19             data: dataNotification
20         };
21
22         fcm.send(message, (err, response)=>{});
23     }
24 };
```

Come visibile, il modulo rende disponibile un oggetto che presenta due metodi.
Il metodo *init* crea un'istanza del modulo *fcm-node*, con la key server fornita dalla console di Firebase per l'invio delle notifiche.
Il metodo *sendNotification* consente l'invio delle notifiche. Il messaggio da inviare è un oggetto che prevede l'inizializzazione di diverse proprietà. La proprietà *to* rappresenta l'identificatore univoco del device a cui mandare la notifica. L'oggetto *notification* rappresenta la notifica da inviare, ed è composto da diverse proprietà, le quali consentono di specificare: il titolo, il corpo della notifica, l'icona della notifica e il comportamento della notifica in caso di tap su di essa.
Oltre a queste due proprietà, è da segnalare la possibilità di inviare dei dati attraverso la proprietà *data*.
Una volta definito il messaggio, per inviare la notifica, è sufficiente chiamare il metodo *send* di *fcm-node*.
Di seguito è possibile notare il comportamento del server ogni qualvolta che riceve una richiesta HTTP per l'invio delle notifiche.

```
1
2  var notification=require('./notification');
3  notification.init('<SERVER_KEY>');
4
5  app.post('/data/send-notification',(req,res)=>{
6      controller.send_notification(req.body.token,req.body.title,
           req.body.body,(rescode,data)=>{
7        res.status(rescode).send(data);
8      })
9  });
10
11 var controller={
12   model.send_notification(token,title,body,()=>{
13     result(201,{});
14   })
15 }
16
17 var model={
18   send_notification: (token,title,body,callback)=>{
19     notification.sendNotification(token,title,body);
20     callback();
21   }
```

```
22 }
```

Prima di definire la route per l'invio delle notifiche, si richiede il modulo illustrato poc'anzi, e lo si inizializza inserendo la server key ottenuta tramite la console di Firebase.
Ogni qualvolta che il server riceve una richiesta per l'invio delle notifiche, effettua una chiamata sul metodo *sendNotification*, recuperando i parametri necessari dal body della richiesta HTTP. Successivamente si invia una risposta alla richiesta HTTP con status code 201 e senza alcun payload.

5.4.4 Generazione regole

Ogni qualvolta che il medico aggiunge, modifica o elimina una regola per un determinato paziente, è necessario modificare il file delle regole del paziente.
Per la generazione del file contenente le regole per un determinato paziente, è nostra intenzione utilizzare il programma Prolog che consente di generare, a sua volta, un programma Prolog contenente le regole per effettuare il rilassamento dei vincoli e con le regole definite dal medico. Pertanto, NodeJs deve essere in grado di eseguire un file Prolog. Il modulo *swipl* fornisce un'interfaccia attraverso la quale è possibile eseguire programmi Prolog.
Di seguito è presente il comportamento del server nel caso in cui il medico aggiunga una regola per un dato paziente.

```
1  app.post('/data/add-rule',(req,res)=>{
2      controller.add_rule(req.body.parameter,req.body.solution,
           req.body.name,req.body.patient,(rescode,data)=>{
3        res.status(rescode).send(data);
4      });
5  });
6
7  var controller={
8    add_rule:(parameter,solution,name,patient,result)=>{
9      model.add_rule(parameter,solution,name,patient,(res)=>{
10       if(res.id===undefined){
11         result(400,{});
12       }else{
13         model.create_prolog_rule(patient,(res2)=>{
14           if(res2===true){
15             result(201,res);
16           }else{
```

```
17              result(400,{});
18            }
19          });
20        }
21      })
22    }
23  }
24
25  var model={
26    add_rule:(parameter,solution,name,patient,callback)=>{
27      rethink.db(db_name).table('rule').insert([{
28        patient,
29        parameter,
30        solution,
31        name
32      }]).run(connection,(err,resultInsert)=>{
33        if(err){
34          callback({id:undefined});
35          return;
36        }
37
38        callback({id:resultInsert.generated_keys[0]});
39      });
40    },
41
42    create_prolog_rule:(patientname,callback)=>{
43      get_rule(patientname,(result)=>{
44        if(result.data.length>0) {
45          var ret = swipl.call('createPrologFile(\'' +
                patientname + '.pl\')');
46          ret = swipl.call('writefact(\'' + patientname + '.pl
                \',\'meal(medium,2).\')');
47          ret = swipl.call('writefact(\'' + patientname + '.pl
                \',\'meal(low,1).\')');
48          ret = swipl.call('writefact(\'' + patientname + '.pl
                \',\'meal(high,3).\')');
49          ret = swipl.call('writefact(\'' + patientname + '.pl
                \',\'lifestyle(low,1).\')');
50          ret = swipl.call('writefact(\'' + patientname + '.pl
                \',\'lifestyle(medium,2).\')');
51          ret = swipl.call('writefact(\'' + patientname + '.pl
                \',\'lifestyle(high,3).\')');
52
53          for (var index in result.data) {
54            var body = '';
```

```
55            var arrayParameter = result.data[index].parameter;
56            var arraySolution = result.data[index].solution;
57
58            for (var i in arrayParameter) {
59              body += arrayParameter[i].parameter + '>=' +
                    arrayParameter[i].min + ',' + arrayParameter[i].
                    parameter + '=<' + arrayParameter[i].max + ',';
60            }
61
62            body += '\nmeal(M,Im),lifestyle(L,Il),\n';
63            body += 'Im>=MinM,Im=<MaxM,Il>=MinL,Il=<MaxL,\n(';
64            for (var i in arraySolution) {
65              body += '(Im>=' + arraySolution[i].minMeal + ',Im=<
                    ' + arraySolution[i].maxMeal + ',Il>=' +
                    arraySolution[i].minActivity + ',Il=<' +
                    arraySolution[i].maxActivity + ');';
66            }
67            body = body.substring(0, body.length - 1);
68            body += '),\nX=[M,L].\n\n';
69
70            ret = swipl.call('addRule(\'' + patientname + '.pl
                  \',\'checkValue\',[\'Glycemia\',\'Alpha\',\'Beta
                  \',\'Meal\',\'Activity\'],[\'MinM\',\'MaxM\',\'
                  MinL\',\'MaxL\'],\'X\',\'' + body + '\')');
71          }
72          callback(true);
73        }else{
74          fs.unlink('..\\prolog\\'+patientname+'.pl', (err) => {
75            if (err){
76              callback(false);
77              return;
78            }
79            callback(true);
80          });
81        }
82      });
83    }
84  }
```

Ogni qualvolta che il server riceve una richiesta per aggiungere una regola, effettua una query per l'inserimento di quest'ultima nel documento relativo alle regole, recuperando dal body della richiesta HTTP le seguenti informazioni:

- Il paziente a cui è associata la regola

- Il nome della regola
- I vincoli della soluzione
- Le condizioni in cui la regola dovrà elaborare un feedback

Una volta inserito con successo la nuova regola definita dal medico per un dato paziente, si recuperano dal database tutte le regole definite per quest'ultimo. Successivamente, si crea il file di regole Prolog. Se il file non esiste (pertanto non era presente alcuna regola), ne verrà creato uno nuovo, altrimenti verrà sovrascritto. Si è scelto di utilizzare questo approccio in quanto, per modificare o eliminare delle regole, poiché tutte presentano il medesimo nome, non sarebbe possibile identificarle nel file Prolog al fine di modificarle o eliminarle. Pertanto, per ogni operazione che prevede l'inserimento, modifica o eliminazione, si ricrea l'intero programma con tutte le regole presenti per quel paziente.
Una volta recuperate le regole, si prosegue, attraverso l'interfaccia *swipl*, con la creazione di un nuovo file; il quale avrà lo stesso nome dello username del paziente a cui sono associate le regole. Una volta creato il file, si prosegue aggiungendo i seguenti fatti:

```
1 meal(low,1).
2 meal(medium,2).
3 meal(high,3).
4 lifestyle(low,1).
5 lifestyle(medium,2).
6 lifestyle(high,3).
```

Creato il file con le regole necessarie per effettuare il rilassamento dei vincoli e, aggiunti i fatti necessari per produrre un feedback, si prosegue con l'inserimento delle regole definite dal medico per il particolare paziente. Per l'inserimento si chiama la regola *addRule*, costruendo la lista dei parametri, la lista dei vincoli e definendo il contenuto della regola che il programma Prolog andrà a generare. Per la costruzione del corpo della regola, si segue la struttura definita nella sezione precedente.
In caso di modifica o eliminazione di una regola, la procedura è analoga a quella appena illustrata. L'unica differenza si può riscontrare nella prima query effettuata sul database. Anziché effettuare una query di inserimento, si effettua una query di update o di delete.

5.4.5 Download regole

Dal flusso di interazioni presentato nell'architettura logica, emerge che l'applicazione Android, in seguito all'autenticazione da parte del paziente, richiede al server il file di regole relativo al paziente. Di seguito è presente l'implementazione di ciò che accade ogni qualvolta che il server riceve la richiesta appena descritta.

```
app.get('/data/get-rule-file', function(req, res){
    controller.get_rule_file(req.query.patients,(filename)=>{
      if(filename!==undefined){
        res.download(filename);
      }else{
        res.status(400).send({});
      }
    })
});

var controller={
  get_rule_file:(patient,result)=>{
    model.get_rule_file(patient,(filename)=>{
      result(filename);
    })
  }
}

var model={
  get_rule_file: (patients,callback)=>{
    var file = '../prolog/'+patient+'.pl';
    if (fs.existsSync(file)) {
      callback(file);
    }else{
      callback(undefined);
    }
  }
}
```

Ogni qualvolta che il server riceve la richiesta, si verifica l'esistenza del file. Si ricorda che il nome del file corrisponde allo username del paziente; il quale viene recuperato dalla query string della richiesta HTTP GET. Se il file è assente, non è stata definita alcuna regola. Pertanto, si restituisce una risposta con status code 400 e senza alcun payload. Altrimenti, in caso di presenza del file, si invia una risposta con un payload, in byte, del file Prolog.

5.4.6 Invio dati simulazione

Ogni qualvolta che il paziente effettua una simulazione, al termine di quest'ultima, l'applicazione Android invia i dati prodotti al server, il quale li memorizza al fine di consentire al medico la visualizzazione di quest'ultimi.

```
app.post('/data/send-data-simulation',(req,res)=> {
  controller.insert_simulation(req.body.data,req.body.username,
     req.body.date,(rescode,data)=>{
    res.status(rescode).send(data);
  })
});

var controller={
  insert_simulation: (data_simulation,username,date,result)=>{
    model.insert_simulation_info(data_simulation,username,date
       ,()=>{})
    model.insert_simulation(username,date,()=>{})
    result(201,{});
  }
}

var model={
  insert_simulation: (username,date,callback)=>{
    rethink.db(db_name).table('simulation').insert([{
      id:username+'_'+date,
      username,
      date
    }]).run(connection,(error)=>{
      if(error) throw error;
    });
    callback();
  },

  insert_simulation_info: (data_simulation,username,date,
     callback)=>{
    var data=[];
    var count=0;
    for(var index in data_simulation) {
      count++;
      data.push({
        id: username + '_'+date + '_' + data_simulation[index].
           step,
        username,
        date,
```

```
36          step: data_simulation[index].step,
37          glycemia: data_simulation[index].glycemia,
38          insulin: data_simulation[index].insulin,
39          glucagon: data_simulation[index].glucagon,
40          meal:data_simulation[index].meal,
41          calories:data_simulation[index].calories
42        });
43      }
44
45      rethink.db(db_name).table('simulation_info').insert(data).
           run(connection,err=>{
46        if(err) throw err;
47      });
48
49      callback()
50    }
51  }
```

Ogni qualvolta che il server riceve la richiesta, recupera dal body di quest'ultima i dati della simulazione che sono contenuti all'interno di un array. Si ricorda che sono presenti nell'array le informazioni di sette giorni, minuto per minuto, per un totale di 10080 elementi. Per ognuno di essi si estraggono le informazioni a cui siamo interessati e, si inseriscono, all'interno del documento *simulation_info*. Successivamente, si inseriscono all'interno del documento *simulation*, alcune informazioni sommarie, quali: username del paziente e la data in cui è stata effettuata la simulazione.

5.4.7 Invio dati sessione

Il paziente, quando effettua una sessione real time, ogni minuto invia una richiesta HTTP POST al server, la quale viene processata nel seguente modo.

```
1  app.post('/data/send-data',(req,res)=>{
2    controller.send_data(req.body,(rescode,data)=>{
3      res.status(rescode).send(data);
4    });
5  });
6
7  var controller={
8    send_data:(data,result)=>{
9      if(data.new){
10       model.insert_new_session(data.username,data.date,(res)=>{
11         if(res===true){
```

```
12            var count=-1;
13            setInterval(()=>{
14              model.session_is_alive(data.username,data.date,
                    count,resultListener=>{
15                if(resultListener==='end'){
16                  clearInterval(this);
17                }
18              });
19            }, 70000);
20          }
21        })
22      }else if(data.end){
23        model.session_end(data.id_session,()=>{
24          result(201,{});
25        });
26      }
27
28      if(data.end==undefined){
29        model.insert_data_session(data,(res)=>{
30          if(res===false){
31            result(400,{});
32          }else if(res===true){
33            result(201,{});
34          }
35        });
36      }
37    }
38  }
39
40  var model={
41    insert_data_session:(data,callback)=>{
42      rethink.db(db_name).table('info').insert([{
43        id:data.username+'_'+data.date,
44        id_session:data.id_session,
45        username:data.username,
46        time: new Date(data.date.replace(' ','T')),
47        glycemia:data.glycemia,
48        insulin:data.insulin,
49        glucagon:data.glucagon,
50        liver_glycogen:data.liver_glycogen,
51        muscle_glycogen:data.muscle_glycogen,
52        calories:data.calories,
53        meal:data.meal
54      }]).run(connection,err=>{
55        if(err){
```

```
56          callback(false);
57          return;
58        }
59        callback(true);
60      });
61    },
62
63    insert_new_session:(username,date,callback)=>{
64      rethink.db(db_name).table('session').insert([{
65        id:username+'_'+date,
66        username,
67        start_time:new Date(date.replace(' ','T')),
68        end_time:new Date(date.replace(' ','T'))
69      }]).run(connection,err=>{
70        if(err){
71          callback(false);
72          return;
73        }
74
75        callback(true);
76      });
77    },
78
79    session_is_alive:(username,date,count,callback)=>{
80      rethink.db(db_name).table('info').filter({id_session:
          username+'_'+date}).count().run(connection,(err,
          countData)=>{
81        if(countData!=count){
82          count=countData;
83          callback('alive');
84        }else{
85          rethink.db(db_name).table('session').filter(rethink.row
              ('id').eq(username+'_'+date).and(rethink.row('
              end_time').eq(rethink.row('start_time')))).update({
              end_time:new Date()}).run(connection,(error)=>{
86            if(error) throw error;
87          });
88          callback('end');
89        }
90      });
91    },
92
93    session_end:(id_session,callback)=>{
94      rethink.db(db_name).table('session').filter({id: id_session
          }).update({end_time: new Date()}).run(connection,(error,
```

```
        resultQuery)=>{
95      if(error) throw error;
96
97      callback();
98    });
99  }
100 }
```

Come si evince dal listato, in base alla presenza di alcuni dati nel body della richiesta, vengono effettuate determinate operazioni.
Se nella richiesta è presente il campo *new*, allora il paziente ha appena iniziato la sessione (ed è trascorso un minuto). In tal caso si effettua una query di inserimento per il documento *session*, in cui si inserisce l'orario di inizio e il paziente che sta effettuando la sessione. Viene inoltre lanciato un task che, ogni 70 secondi, controlla nel documento relativo ai dettagli della sessione, se il numero di informazioni inviate dal paziente, per quella relativa sessione, siano cambiati o meno. Se il numero di informazioni risulta essere lo stesso, a distanza di 70 secondi, si può dedurre che la comunicazione tra il server e l'applicazione è interrotta. In tal caso, si registra la sessione come terminata.
Se nella richiesta è presente il campo *end*, allora il paziente ha terminato la sessione. Pertanto, si effettua una query di update nel documento che tiene traccia delle sessioni effettuate dai pazienti.
In caso di assenza del campo *end*, si inserisce nel documento che tiene traccia dei dettagli delle sessioni, i dati relativi alla sessione per il minuto in cui è stata inviata la richiesta.

5.4.8 Server Sent Events

Se il medico curante vuole visualizzare i dati della sessione real time mentre quest'ultima è in corso, per evitare che egli effettui continuamente il refresh della pagina, è necessario definire lato server il supporto ai Server Sent Events (SSE). Di seguito è possibile trovare l'implementazione per l'invio dei dati relativi alla sessione di un particolare paziente.

```
1 app.get('/data/*/session',(req,res)=>{
2   controller.session_sse(req,res,()=>{});
3 });
4
5 var controller={
6   session_sse:(req,res,result)=>{
```

```
7       model.session_sse(req,res,()=>{
8         result();
9       });
10    }
11  }
12
13  var model={
14    session_sse:(req,res,callback)=>{
15      req.socket.setTimeout(Infinity | 0);
16      res.writeHead(200, {
17        'Content-Type': 'text/event-stream',
18        'Cache-Control': 'no-cache',
19        'Connection': 'keep-alive'
20      });
21
22      var usernameUrl=req.originalUrl.substring(1).split('/')[0];
23
24      rethink.db('diabetes_monitoring').table('session').filter({
            username:usernameUrl}).changes().run(connection,function
            (err5, cursor) {
25        cursor.each((_,value)=>{
26          res.write('data: '+JSON.stringify(value.new_val)+'\n\n'
                );
27        });
28        req.on('close', function() {
29          cursor.close();
30        });
31      });
32
33      callback();
34    }
35  }
```

Il server processa tutte le richieste in cui l'url ha una struttura del tipo */data/*/session*; dove "*" può essere una qualsiasi stringa.
Ogni qualvolta che il server riceve una richiesta HTTP GET di questo tipo, si imposta il timeout della richiesta a infinito. Inoltre si definisce l'header della risposta alla richiesta.
Dall'url si recupera lo username del paziente e, si chiede al database, di essere notificati ogni qualvolta che nel documento *session* vi è un cambiamento nei record in cui lo username del paziente coincide con quello recuperato dall'url. Poiché l'unico cambiamento che può presentarsi in questo documento è un inserimento, siamo certi che, ogni qualvolta che verrà effettuata la callback, il

server avrà appena ricevuto dei dati relativi alla sessione del paziente. Pertanto, ogni qualvolta che viene invocata la callback, si inviano, in formato stringa, i dati del minuto corrente.

5.5 Struttura interfaccia web

L'interfaccia web che consente ai medici di accedere e monitorare la situazione dei propri assistiti è realizzata con il framework Angular.
All'interno di questa sezione verranno presentate le funzionalità attraverso le quali il medico curante può monitorare i propri pazienti. Poiché non si tratta di un aspetto fondamentale ai fini della tesi, verranno mostrati gli aspetti implementativi di rilevanza.

5.5.1 Bootstrap applicazione web

Come è stato detto nel paragrafo in cui sono state presentate le tecnologie selezionate, in Angular, per realizzare un'applicazione web, è necessario definire il *root module*; ovvero la componente che contiene tutte le informazioni necessarie per il bootstrap dell'applicazione web. Di seguito la struttura del modulo definito per il caso di studio.

```
...

@NgModule({
  declarations: [
    AppComponent,
    LoginComponent,
    PatientListComponent,
    AlertUsernameComponent,
    PatientComponent,
    ListSimulationSessionComponent,
    DetailComponent,
    NotificationComponent,
    PressureGlucoseComponent,
    RulesListComponent,
    DetailRuleComponent,
    AddRuleComponent
  ],
  imports: [
    MatListModule,
    MatDatepickerModule,
```

```
21      MatNativeDateModule,
22      MatFormFieldModule,
23      HttpClientModule,
24      BrowserAnimationsModule,
25      MatDialogModule ,
26      FormsModule,
27      BrowserModule,
28      routing
29    ],
30    providers: [ Global,DateFormat ],
31    bootstrap: [AppComponent],
32    entryComponents: [AlertUsernameComponent,
          NotificationComponent]
33  })
34
35  ...
```

Nella proprietà *declarations* sono presenti i componenti che costituiscono l'applicazione web. Come detto nel paragrafo relativo alla presentazione di Angular, i component gestiscono la logica applicativa e i dati, rendendoli disponibili al template.
Attraverso la proprietà *imports* si definiscono i servizi necessari all'applicazione web.
All'interno della proprietà *providers* si definiscono i servizi del provider. Il servizio *Global* espone le proprietà e le funzioni di utility necessarie per il corretto funzionamento dell'applicazione web, mentre il servizio *DateFormat* consente di effettuare il parsing di una stringa a un oggetto *Date*. La proprietà *entryComponents* permette di definire dei componenti che Angular inizializza in modo imperativo; ovvero senza dover far riferimento ad essi all'interno del modello.
Il component *AppComponent* definito all'interno della proprietà *bootstrap*, verrà utilizzato al momento del bootstrap dell'applicazione.
L'oggetto *routing*, presente all'interno della proprietà *imports*, contiene le rotte delle pagine dell'applicazione. Al fine di migliorare la leggibilità del modulo, quest'ultime sono state definite all'interno dell'oggetto *routing*. Di seguito sono presenti l'insieme delle rotte definite per l'applicazione web.

```
1
2  const routes: Routes = [{
3    path: '',
4    component: LoginComponent
5  },
```

```
6     {
7       path: 'patient-list',
8       component: PatientListComponent
9     },
10    {
11      path: 'detail-rule',
12      component: DetailRuleComponent
13    },
14    {
15      path: 'add-rule',
16      component: AddRuleComponent
17    },
18    {
19      path: 'rules-list',
20      component: RulesListComponent
21    },
22    {
23      path: 'patient',
24      component: PatientComponent
25    },
26    {
27      path: 'list-data',
28      component: ListSimulationSessionComponent
29    },
30    {
31      path: 'detail',
32      component: DetailComponent
33    },
34    {
35      path: 'log',
36      component: PressureGlucoseComponent
37    }];
38
39  export const routing: ModuleWithProviders = RouterModule.
      forRoot(routes);
```

Come si evince dal listato, si definisce un array in cui ogni elemento è costituito da un oggetto con due proprietà: *path* e *component*. *Path* rappresenta l'uri relativo alla pagina, mentre il parametro *component* definisce chi gestirà la logica applicativa, i dati e sarà collegato alla pagina HTML che verrà visualizzata.
Nel *root module* mostrato poc'anzi, all'interno della proprietà *bootstrap*, è stato definito il component da utilizzare al momento bootstrap dell'applicazione.

Nel listato che segue, è presente la sua implementazione.

```
  @Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent{
  constructor(public globalVariable: Global,private router:
     Router){
  }

  private logout():void{
    this.globalVariable.setUsername('');
    sessionStorage.patient=undefined;
    sessionStorage.dataSimulationSession=undefined;
    sessionStorage.token=undefined;
    sessionStorage.page=undefined;
    this.router.navigateByUrl('/');
  }
}
```

Il component, attraverso il decoratore, definisce: l'uri per visualizzare la pagina, il percorso della pagina HTML associato e lo stile da applicare alla pagina. La classe espone un metodo *logout*, il quale, come intuibile dal nome, si occupa di gestire il logout dell'utente; eliminando le variabili di sessione ed effettuando il reindirizzamento alla schermata di login.
Nel listato sottostante, è presente la pagina HTML associata all'AppComponent.

```
  <header class="navbar navbar-default mynavbar" id="mynavbar"
     *ngIf="this.router.url !== '/'">
  <div class="row">
    <div class="col-md-10 divTitle centered"><span> e-Diabetes
       </span></div>
    <div (click)="logout()" class="col-md-2 myicon centered
       myiconUnselected"><span class="glyphicon glyphicon-log-
       out myicon"><span> Logout</span></span></div>
  </div>
</header>

<router-outlet></router-outlet>
```

Come visibile, è presente un tag definito da Angular: *router-outlet*. Questo tag permette di renderizzare al suo interno le pagine HTML degli altri component.

Attraverso la direttiva strutturale *ngIf*, quando l'utente si troverà in una pagina diversa dalla schermata di login, verrà renderizzato e mostrato un header; attraverso il quale sarà possibile effettuare il logout.

5.5.2 Login/Registrazione

La prima pagina che viene visualizzata all'apertura dell'applicazione web è la schermata di login; attraverso la quale è possibile effettuare l'accesso o creare una nuovo utente.
Al momento della registrazione, il server genera uno username e, alla ricezione di quest'ultimo, viene mostrato un pop-up contenente lo username necessario per effettuare l'accesso.
In caso di registrazione, si invia al server la richiesta presente nel listato sottostante.

```
this.http.post('http://localhost:3000/user/new-user', JSON.
    stringify({
    physician:true,
     password: this.password,
     first_name: this.first_name,
     last_name: this.last_name
})).subscribe(data=>{
     this.globalVariable.setUsername(data.username);
     this.openDialog('Your username will be: '+data.username);
})
```

Attraverso il servizio nativo di Angular, è possibile inviare delle richieste HTTP. Nel caso della registrazione, si invia una richiesta POST; il cui body è rappresentato da un oggetto JSON contenente le informazioni necessarie. Alla ricezione di una risposta da parte del server, la callback si limita a mostrare, attraverso un pop-up, lo username assegnato al medico.
In caso di login, si invia al server la richiesta presente nel listato sottostante.

```
this.http.post('http://localhost:3000/user/login', JSON.
    stringify({
     username:this.globalVariable.getUsername(),
     password: this.password
})).subscribe(data=>{
     this.router.navigateByUrl('/patient-list')
},err=>{
     if(err.status===401){
```

```
8           this.openDialog('Username or password are invalid')
9       }
10  })
```

Come per la registrazione, si invia una richiesta POST al server. Il contenuto del body della richiesta è costituito dallo username e dalla password inserite dal medico. In caso di esito positivo da parte del server, la callback si occuperà di indirizzare l'utente alla pagina in cui verrà mostrata la lista dei pazienti; altrimenti un pop-up notificherà il mancato accesso.

5.5.3 Lista pazienti

Una volta effettuato il login, al medico viene mostrata la lista dei pazienti. All'apertura della pagina, si invia una richiesta GET al server, attraverso la quale si recuperano gli assistiti del medico.
Cliccando su un paziente, è possibile visualizzare una schermata di dettaglio e, attraverso di essa, il medico può effettuare le seguenti azioni:

- Visualizzare lo storico delle misurazioni relative alla pressione arteriosa
- Visualizzare lo storico delle misurazioni relative alla glicemia
- Visualizzare lo storico delle sessioni real time
- Visualizzare lo storico delle simulazioni
- Gestire il sistema di regole definito per l'assistito

Di seguito è presente il listato in cui si effettua la richiesta al server per ottenere la lista degli assistiti.

```
1  this.http.get('http://localhost:3000/user/get-user-list?
       physician=' + this.globalVariable.getUsername()).subscribe(
       data => {
2    for (var index in data) {
3        var dateString = data[index].birth_date.split('-');
4          var date = new Date((dateString[2].split('T'))[0],
               dateString[0], dateString[1]);
5          var ageDifMs = Date.now() - date.getTime();
6          var ageDate = new Date(ageDifMs);
7          var age = Math.abs(ageDate.getUTCFullYear() - 1970);
```

```
8
9           this.patients.push({
10            age: age,
11            cf: data[index].cf,
12            first_name: data[index].first_name,
13            last_name: data[index].last_name,
14            gender: data[index].gender,
15            id: data[index].id,
16            token: data[index].token
17          });
18      }
19  })
```

Trattandosi di una GET, i parametri si inseriscono all'interno della query string. Alla ricezione di una response positiva da parte del server, si recupera l'oggetto JSON e si itera l'array per recuperare la lista dei pazienti. Per ogni paziente si estraggono le informazioni a esso associate, quali: data di nascita, codice fiscale, nome, cognome, sesso e token necessario per l'invio della notifica push attraverso Firebase.
Ottenuta la data di nascita, si calcola l'età e, successivamente, si creano degli oggetti da inserire all'interno di un array. Attraverso il binding, l'array viene impiegato per mostrare l'elenco degli assistiti al medico.

```
1   <mat-list>
2   <h1>Patients</h1>
3   <mat-list-item *ngFor="let patient of patients; let last=last
       " (click)="openPage(patient)">
4     <div mat-line><span>Name:</span> {{patient.first_name}} {{
         patient.last_name}}</div>
5     <div mat-line><span>CF:</span> {{patient.cf}}</div>
6     <div mat-line><span>Gender:</span> {{patient.gender}} Age:
         {{patient.age}}</div>
7     <mat-divider *ngIf="!last"></mat-divider>
8   </mat-list-item>
9   </mat-list>
```

La direttiva strutturale **ngFor* consente di replicare un elemento *n* volte. Pertanto, una volta ottenuta la lista dei pazienti, il numero di elementi *mat-list-item* sarà uguale alla lunghezza dell'array *patients*.
Gli elementi *mat-list*, *mat-list-item*, *mat-line* e *mat-divider* sono nativi di Angular e forniscono uno stile material alla pagina, ma senza definire alcun comportamento.

5.5.4 Gestione regole

Il medico, attraverso l'applicazione web, può visualizzare e gestire le regole definite per un particolare paziente.
Definita una nuova regola, si invia una richiesta POST al server; affinché quest'ultimo generi il nuovo file Prolog contenente la regola appena definita.
Per eliminare o modificare una regola, si invia una richiesta POST al server, il quale si occupa di rigenerare il file Prolog.
L'elenco delle regole definite dal medico, per un particolare paziente, si recuperano attraverso la seguente richiesta HTTP GET.

```
this.http.get('http://localhost:3000/data/get-rule?physician='
   + this.globalVariable.getUsername()+'&patient='+json.id).
   subscribe(data => {
this.rules=data.data;
});
```

Come parametri della query string si impostano lo username del medico e lo username del paziente per il quale si vogliono visualizzare le regole. In caso di responso positivo da parte del server, si memorizza il payload che corrisponde a un array di oggetti. Ogni elemento contiene l'id che viene assegnato dal database per ogni documento. Tale id verrà utilizzato in caso di modifica o eliminazione della regola da parte del medico.
Per la creazione di una regola, ogni qualvolta che il medico aggiunge una condizione per la quale la regola dovrà analizzare i dati, o un vincolo sulla soluzione che dovrà essere prodotta, si esegue, a seconda dei casi, una delle seguenti operazioni:

```
  private addParameterConstraint():void{
    this.dataParameters.push({
      min:undefined,
      parameter:'',
      max:undefined
    });
  }

  private addSolutionConstraint():void{
    this.dataSolution.push({
      minMeal:'',
      maxMeal:'',
      minActivity:'',
      maxActivity:''
    });
```

```
16   }
```

Quando il medico aggiunge una condizione o un vincolo, si aggiunge nel relativo array un oggetto inizialmente vuoto. Attraverso Angular si effettua il binding con l'array e, ogni qualvolta che si modifica un valore attraverso il DOM, si modifica l'oggetto del relativo elemento dell'array.
Quando il medico aggiunge una regola, si invia la seguente richiesta POST al server.

```
1  var json = JSON.parse(sessionStorage.patient);
2      this.http.post('http://localhost:3000/data/add-rule',
           JSON.stringify({
3      patient:json.id,
4      name:this.name,
5      parameter:this.dataParameters,
6      solution:this.dataSolution
7    }), { headers: new HttpHeaders().set('Content-Type', '
       application/json')}).subscribe(data=>{
8        this.router.navigateByUrl('/rules-list');
9      })
```

Il body della richiesta consiste in un JSON che contiene: lo username del paziente per il quale aggiungere la regola, il nome della regola, i parametri e i vincoli della soluzione della regola.
In caso di responso positivo da parte del server, si procede a reindirizzare il medico verso la pagina contenente la lista delle regole per il paziente selezionato.
Quando il medico modifica una regola, a differenza dell'url, si procede nel medesimo modo.
Quando il medico vuole eliminare una regola, invece, si invia la seguente richiesta POST al server.

```
1  var json = JSON.parse(sessionStorage.patient);
2    this.http.post('http://localhost:3000/data/delete-rule',
       JSON.stringify({
3      id:this.id,
4      patient:json.id
5    }), { headers: new HttpHeaders().set('Content-Type', '
       application/json')}).subscribe(data=>{
6        this.router.navigateByUrl('/rules-list');
7      })
```

Il body della richiesta è rappresentato da un oggetto contenente l'id della regola assegnata dal database e dallo username del paziente.

5.5.5 Server Sent Events

I Server Sent Events(SSE) consentono a una pagina web di ricevere degli aggiornamenti dal server, senza il bisogno di inviare una richiesta a quest'ultimo. A differenza delle web-socket, la comunicazione con i SSE è unidirezionale.
I SSE vengono utilizzati, all'interno di questa interfaccia web, per aggiornare l'elenco delle simulazioni/sessioni effettuate dal paziente e per la ricezione dei dati nel corso di una sessione real time.
Di seguito il dettaglio dell'implementazione.

```
var source = new EventSource('http://localhost:3000/data/' +
    id_session + '/stream')
              source.onmessage = (event) => {
                var json = JSON.parse(event.data)
                var stringTime = json.time.replace('T', ' ').
                    replace(/-/g, ' ').split(' ')

                this.data.push({
                  calories: json.calories,
                  glucagon: json.glucagon,
                  glycemia: json.glycemia,
                  insulin: json.insulin,
                  liver_glycogen: json.liver_glycogen,
                  muscle_glycogen: json.muscle_glycogen,
                  time: stringTime[2] + '/' + stringTime[1] + '
                      /' + stringTime[0] + ' ' + stringTime[3].
                      split('.')[0],
                  meal: json.meal
                })

                this.zone.run(() => {
                  this.data = this.data.slice(0, this.data.
                      length);
                })
              }
```

L'EventSource è un'istanza che consente di stabilire una connessione persistente con un server HTTP verso l'indirizzo specificato nel costruttore.

Una volta creata l'istanza, si assegna alla proprietà *message*, l'event handler che si occupa di gestire la ricezione dei messaggi da parte del server. In questo specifico caso, si gestisce la ricezione dei dati relativi a una sessione real time.

5.5.6 Invio notifiche

L'invio delle notifiche è possibile solo nei seguenti casi:

- Visualizzazione misure pressione di un paziente
- Visualizzazione misure glicemia di un paziente
- Visualizzazione lista simulazioni/sessioni di un paziente
- Visualizzazione dettagli simulazioni/sessioni di un paziente

Di seguito l'implementazione per l'invio delle notifiche.

```
1  this.http.post('http://localhost:3000/data/send-notification'
       , JSON.stringify({
2       token:sessionStorage.token,
3       title:this.title,
4       body:this.body
5     }), { headers: new HttpHeaders().set('Content-Type', '
        application/json')}).subscribe(data=>{
6       })
```

Si invia una richiesta HTTP POST al server. il body della richiesta è costituito dal token del device a cui inviare la notifica, dal titolo della notifica e dal messaggio della notifica.

Capitolo 6

Il progetto: implementazione dell'applicazione Android

L'applicazione Android rappresenta il cuore di tutto il sistema. Attraverso di essa il paziente è in grado di visualizzare e simulare il proprio andamento glicemico nel tempo, ricevendo dei feedback volti a migliorare il proprio stato di salute.

Nel primo paragrafo del capitolo vengono illustrati i framework e le API necessarie per la realizzazione di questa componente del sistema.
Il secondo paragrafo è dedicato al modello utilizzato per simulare il metabolismo del glucosio.
I restanti paragrafi del capitolo sono dedicati agli aspetti implementativi di maggiore rilevanza.

6.1 Framework e API selezionati

6.1.1 API Bluetooth

Android fornisce un supporto nativo per permettere agli sviluppatori di interagire con altri device attraverso la tecnologia Bluetooth; consentendo di realizzare connessioni point-to-point e/o multipoint. L'API fornita da Android consente di:

- Effettuare il discovery dei dispositivi Bluetooth

- Visualizzare i dispositivi Bluetooth con cui si sono già effettuate delle connessioni
- Stabilire dei canali RFCOMM
- Stabilire una connessione Bluetooth attraverso il discovery
- Scambiare dati con il dispositivo con cui si è accoppiati
- Gestire connessioni multiple

L'utilizzo di questa API è necessaria per consentire il pairing e lo scambio di dati tra il device mobile dell'assistito e il body gateway.

6.1.2 Volley

Volley è un'API nativa di Android che consente la trasmissione di dati attraverso la rete. *Volley* offre le seguenti funzionalità:

- Scheduling automatico delle richieste di rete
- Connessioni concorrenti multiple
- Supporto alla prioritizzazione delle richieste
- API per la cancellazione delle richieste
- Customizzazione
- Tool per il debugging

L'utilizzo di questa API è fondamentale per la comunicazione tra il device mobile dell'assistito e il server.

6.1.3 MASON

MASON (Multi-Agent Simulator Of Neighborhoods) è un ambiente di simulazione multi-agente ad eventi discreti, realizzato all'interno della *George Mason University*. Le caratteristiche che contraddistinguono questo ambiente di simulazione sono:

- Livello di presentazione completamente separato dal modello. È possibile eseguire dei modelli senza una UI o, addirittura, è possibile intercambiare più livelli di visualizzazione per un modello.

- I modelli realizzati tramite *MASON* sono serializzabili sul disco. Ad esempio è possibile salvare l'esecuzione del modello in un certo stato e, in seguito, riprendere l'esecuzione dal punto in cui è stato interrotto; sulla stessa macchina o su una diversa.
  È anche possibile interrompere l'esecuzione, cambiare il livello di visualizzazione e riprendere l'esecuzione.

- I modelli sono completamente incapsulati. Lo stesso processo può eseguire in parallelo più modelli *MASON*, oppure, possono essere eseguiti in concorrenza su più thread.

- Realizzato in Java per permettere l'esecuzione dei modelli in ambienti eterogenei

- Utilizzo dell'algoritmo *Mersenne Twister* per la generazione di numeri pseudocasuali

- I risultati prodotti da un modello con una determinata configurazione sono replicabili

- È modulare e consistente. Offre un alto livello di separazione e indipendenza tra i vari elementi del sistema. Permette di usare e ricombinare diverse parti del sistema in svariati modi.

Agente

Con il termine *agente* si indica una qualsiasi entità in grado di percepire l'ambiente in cui si trova e compiere delle azioni su di esso, in modo autonomo, proattivo e reattivo. Gli agenti, oltre a interagire con l'ambiente, sono anche in grado di comunicare tra di loro.
Un sistema formato da più agenti prende il nome di sistema multi-agente (MAS).
La nozione appena data è piuttosto debole per definire un agente. Una nozione più forte di agente prevede che essi siano in grado di pianificare e raggiungere

degli obiettivi, sulla base delle proprie conoscenze, intenzioni e percezione dell'ambiente in cui sono situati.

Architettura

Come già accennato, una delle caratteristiche di *MASON* è la separazione che c'è tra il livello di presentazione e il modello. Di seguito è presente un'immagine che illustra l'architettura di *MASON*.

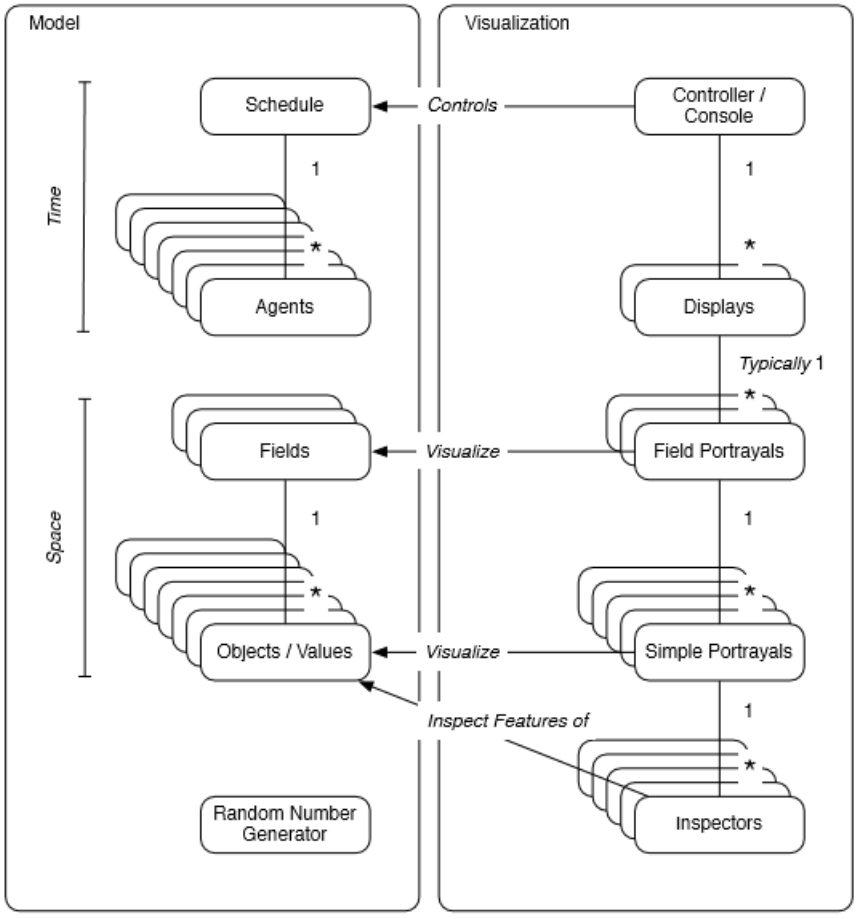

Figura 6.1: Architettura MASON[17]

MASON fornisce gli strumenti necessari per la visualizzazione del modello, sia in 2D che in 3D, incapsulati nella classe *GUIState*. Tale classe contiene un oggetto controller, rappresentato da una console, attraverso la quale è possibile avviare/fermare e manipolare lo scheduler del modello.
Il modello è interamente incapsulato nella classe *SimState*. Tale classe contiene un oggetto che agisce da scheduler a eventi discreti, attraverso il quale è possibile definire l'ordine temporale con cui gli agenti dovranno essere eseguiti.
Il principio con cui *MASON* esegue il modello realizzato può essere riassunto nel seguente pseudocodice.

```
1   state <- sottoclasse SimState;
2
3   //associa un nome al thread
4   state.nameThread();
5
6   //numero di volte in cui ripetere la simulazione
7   state.setJob(N);
8
9   //inizio la simulazione
10  state.start();
11
12  //mando avanti la simulazione di uno step alla volta
13  loop(state.schedule.step(state))
14
15  //termine della simulazione
16  state.finish();
```

L'oggetto *schedule* fornito dalla classe *SimState* è uno scheduler a eventi discreti. Si tratta di una struttura dati in cui è possibile definire quali agenti devono essere eseguiti in un certo istante di tempo. Attraverso l'override del metodo *start()* della classe *SimState*, è possibile aggiungere gli agenti allo scheduler, associandogli l'istante di tempo in cui essi dovranno essere eseguiti.
All'invocazione del metodo *step()*, lo scheduler incrementa il proprio tempo di uno e, successivamente, esegue l'azione di tutti gli agenti schedulati per quell'istante di tempo. Quando non sono più presenti agenti nello scheduler, viene invocato il metodo *finish()*, che pone fine alla simulazione.
Un agente, per essere eseguito dallo scheduler, deve implementare l'interfaccia *Steppable*. Attraverso il metodo *step()* definito dall'interfaccia, è possibile definire l'azione che l'agente dovrà eseguire.
Vi sono due metodologie per schedulare gli agenti: *one-shot* o *repeating*. Con la metodologia one-shot, l'agente, in base all'istante di tempo definito, viene chiamato una sola volta dallo scheduler, per poi essere eliminato al termine del-

l'esecuzione. Con la metodologia repeating, invece, lo scheduler esegue l'azione dell'agente a ogni step.

6.1.4 JIProlog

Il sistema di regole che si occupa di fornire un feedback al paziente, come visto nel capitolo precedente, è stato realizzato sotto forma di programma Prolog. Siccome si vuole eseguire il sistema di regole sul device dell'assistito e, Android non supporta l'esecuzione nativa, è necessario un interprete Prolog realizzato in Java.
JIProlog è una libreria Java che fornisce un interprete Prolog cross-platform e open-source. Le peculiarità di questa libreria sono:

- Possibilità di chiamare predicati Prolog da Java
- Possibilità di invocare metodi Java da Prolog come se fossero predicati
- Possibilità di collegare un database JDBC da Prolog

6.2 Modello diabete

Poiché bisogna simulare l'andamento glicemico del paziente, è necessario utilizzare un modello che sia in grado di simulare il metabolismo del glucosio. A tal fine, si è scelto di adoperare il modello realizzato durante il lavoro di tesi di *Francesco Degli Angeli* [18]. Si tratta di un modello agent-based per il self-Management del diabete mellito di tipo 1, basato su *MASON*.
Il modello consente di simulare il metabolismo di una persona sana, di una persona affetta da diabete mellito di tipo 1 e di una persona affetta da diabete mellito di tipo 1 che assume insulina in concomitanza dei pasti.
All'interno di questo paragrafo è presente una panoramica degli agenti che compongono il modello e il relativo comportamento.
Il paragrafo si conclude con una sezione in cui vengono illustrate le modifiche apportate al modello.

6.2.1 Agenti

Il modello adoperato riproduce il metabolismo del glucosio in condizioni fisiologiche. Le entità che compongono il modello sono:

- **Intestino:** consente di assorbire il glucosio dagli alimenti
- **Sistema circolatorio:** trasporta il glucosio negli organi necessari
- **Pancreas:** produce insulina e glucagone
- **Fegato:** immagazzina glucosio per liberarlo nelle situazioni in cui è richiesto
- **Muscoli:** consumano glucosio
- **Cervello:** consuma circa 120 grammi di glucosio al giorno
- **Paziente:** svolge attività fisica ed effettua dei pasti

Le entità appena descritte sono state modellate come agenti presenti nell'ambiente, mentre il sistema circolatorio rappresenta l'ambiente. Di seguito sono presenti le caratteristiche degli organi modellati.

Intestino

Tale agente assorbe i carboidrati ingeriti e li riduce in glucosio; il quale viene immesso nel sistema circolatorio. L'agente interagisce con l'ambiente per verificare la presenza di alimenti e, immettere nel flusso sanguigno, il glucosio derivato dalla digestione di quest'ultimi.

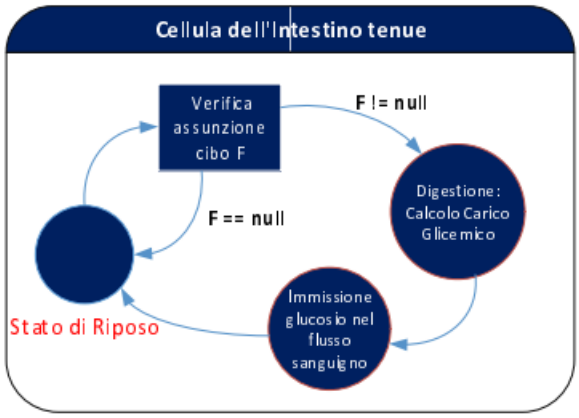

Figura 6.2: Diagramma degli stati dell'intestino[18]

Pancreas

Il pancreas ha il compito di rilasciare l'ormone dell'insulina (cellule β) e del glucagone (cellule α).
Se il livello di glucosio scende al di sotto della soglia dei 75 mg/dl, le cellule *alpha* iniziano a secernere glucagone; il quale stimola il fegato a rilasciare glucosio. Al rientrare della glicemia nei valori di riferimento, la produzione di tale ormone cessa. Queste cellule interagiscono con l'ambiente per controllare che il valore del glucosio non sia inferiore ai 75 mg/dl e per secernere glucagone nel sangue.

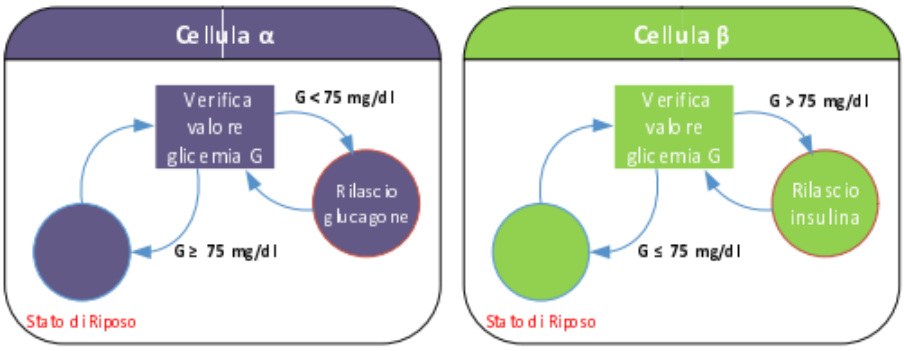

Figura 6.3: Diagramma degli stati del pancreas[18]

Le cellule β rilasciano insulina quando la concentrazione di glucosio è superiore ai 75 mg/dl; per poi cessare la produzione quando la concentrazione scende al di sotto di questa soglia. Queste cellule interagiscono con l'ambiente per verificare la concentrazione di glucosio e per secernere insulina nel sangue.

Fegato

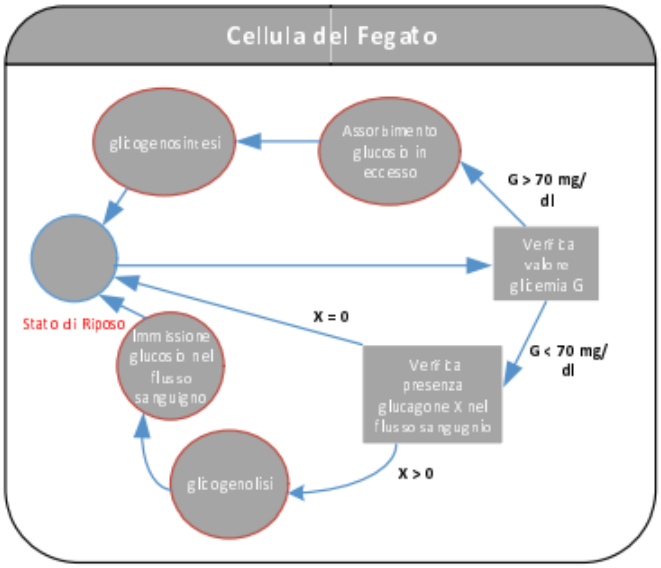

Figura 6.4: Diagramma degli stati del fegato[18]

Il fegato ha il compito di rilasciare, quando è necessario, il glucosio nel flusso sanguigno, oppure assorbirlo quando è in eccesso. Il comportamento è determinato dalla concentrazione di insulina e di glucagone. In presenza di glucagone, il fegato avvia il processo di glicogenolisi, il quale permette di trasformare le riserve di glicogeno in glucosio e rilasciarlo nel sangue. Se il glucosio nel sangue è in eccesso, il fegato immagazzina il glucosio e lo trasforma in glicogeno (glicogenosintesi). Pertanto, tale agente interagisce con l'ambiente per verificare la presenza del glucagone e della concentrazione di glucosio nel sangue. A seconda di questi valori, il fegato determina se rilasciare o assorbire il glucosio.

Muscoli

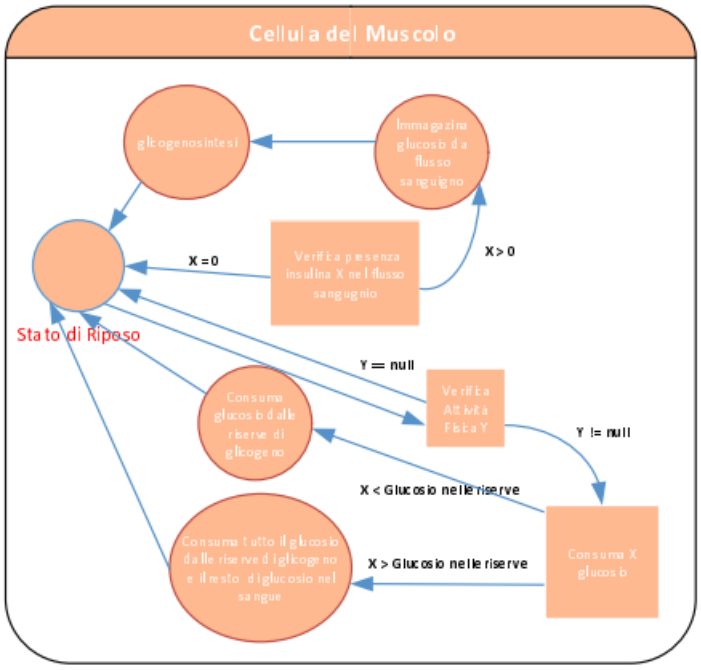

Figura 6.5: Diagramma degli stati dei muscoli[18]

I muscoli, in caso di presenza di insulina nel flusso sanguigno, assorbono il glucosio e lo trasformano in glicogeno attraverso il processo di glicogenesi. In caso di attività fisica, quest'ultimi consumano il glicogeno immagazzinato e, una volta esauritosi, assorbono il glucosio presente nel flusso sanguigno. Pertanto, le interazioni con il flusso sanguigno consistono nel verificare la presenza di insulina e di glucosio. Inoltre, verificano se si sta effettuando attività fisica al fine di consumare il glicogeno immagazzinato.

Cervello

In una giornata il cervello consuma in media 120 grammi di glucosio. L'unica interazione con l'ambiente consiste nell'assorbire il glucosio dal flusso sanguigno.

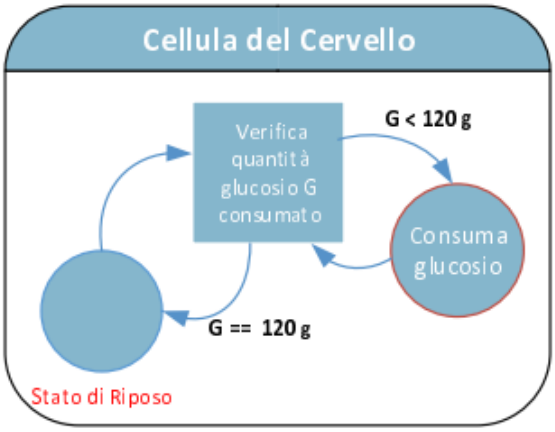

Figura 6.6: Diagramma degli stati del cervello[18]

Paziente

Tale agente si occupa di simulare il comportamento del paziente reale. Attraverso di esso vengono inserite le attività fisiche svolte da quest'ultimo. Inoltre, attraverso questo agente, è possibile inserire i pasti e, in caso di simulazione di diabete di tipo 1, è possibile ottenere la quantità di insulina da iniettare.

6.2.2 Modifiche al modello

Sul modello realizzato da [18], sono necessarie alcune modifiche per consentirne l'utilizzo nel sistema che è nostra intenzione realizzare.
Il modello utilizza delle componenti grafiche che si basano su API Java che non sono presenti in Android. Pertanto, al fine di rendere possibile l'utilizzo del modello in un ambiente Android, è necessaria la rimozione delle componenti grafiche dal modello.
Il sistema oggetto di questa tesi è rivolto a persone affette da diabete, pertanto, nel modello che si intende utilizzare, si rende necessaria la modifica dell'agente che modella il pancreas. In particolare, è necessario rimuovere dal modello la produzione dell'insulina.
Oltre a modificare l'agente che modella il pancreas, si rende necessaria la rimozione dell'agente che modella il paziente. Attraverso questo agente è possibile simulare i pasti e l'attività fisica svolta da un'ipotetica persona. Poiché il nostro sistema prevede un'interazione reale con il paziente, tale componente non

risulta essere necessaria.
In base ai diagrammi di sequenza definiti nel terzo capitolo, si evince che il modello, nel corso della simulazione, scambia alcune informazioni con un controller. Poiché il modello non è predisposto per uno scambio di dati nel corso della simulazione, al fine di permettere ciò, è necessario realizzare la soluzione presente nella figura successiva.
La soluzione è ideata per consentire, a qualsiasi modello MASON, lo scambio di dati nel corso della simulazione.
La classe *Model* estende la classe *SimState* fornita da MASON. I metodi astratti *setData(JSONObject json)* e *getSimulationData()*, rappresentano i metodi attraverso i quali il modello potrà inviare e ricevere informazioni nel corso della simulazione. Poiché si trattano di metodi astratti, dovranno essere implementati dalle sottoclassi.
Il metodo *start(JSON)* accetta un oggetto JSON; il quale può essere utilizzato per passare, al momento dell'avvio della simulazione, i parametri necessari. Il metodo, quando invocato, invoca il metodo *start()* della classe *SimState*; il quale si occuperà di avviare la simulazione.

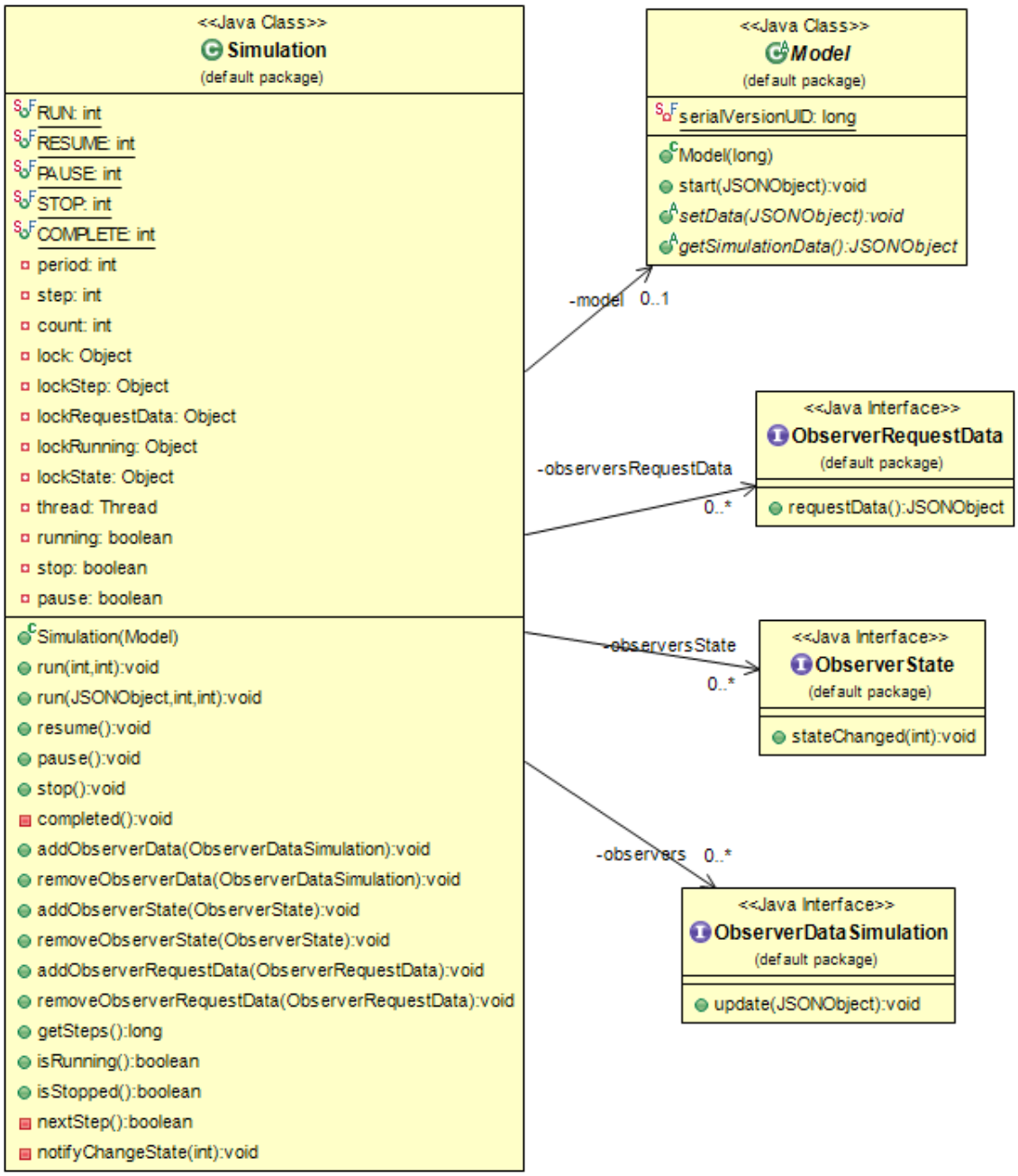

Figura 6.7: Diagramma delle classi per lo scambio dei dati nel corso della simulazione

Di seguito è presente l'override del metodo *start*, descritto poc'anzi, per il nostro caso di studio.

```
@Override
public void start(JSONObject json){
  try{
    this.foods=(ArrayList<Food>)json.get("foods");
    isMale=json.getBoolean("isMale");
    age=json.getInt("age");
    weight=json.getDouble("weight");
    interAt=json.getInt("lifestyle");
    glycemia=json.getDouble("glycemia");
    glucagon=json.getDouble("glucagon");
    insulin=json.getDouble("insulin");
    liver_glycogen=json.getDouble("liverglycogen");
    muscle_glycogen=json.getDouble("muscleglycogen");
    basalInsulin=insulin;

  }catch(JSONException e){
    e.printStackTrace();
  }

  super.start(json);
}

@Override
public void start()
{
  super.start();

  diabetes1 = true;
  selfManagement = true;
  pda = new Pda();

  AlphaCell alphaCell = new AlphaCell();
  LiverCell liverCell = new LiverCell(liver_glycogen);
  BrainCell brainCell = new BrainCell();
  MuscleCell muscleCell = new MuscleCell(muscle_glycogen);
  Decay decayInsulin = new Decay();

  meals=new Meals(foods);
  IntestineCell intestineCell = new IntestineCell();

  schedule.scheduleRepeating(intestineCell);
  schedule.scheduleRepeating(alphaCell);
  schedule.scheduleRepeating(liverCell);
  schedule.scheduleRepeating(muscleCell);
  schedule.scheduleRepeating(brainCell);
  schedule.scheduleRepeating(decayInsulin);
}
```

Come visibile dal listato, sono presenti i metodi *start* definiti dalle classi*Model*

e *SimState*. Nel metodo *start* definito dalla classe *Model*, è possibile notare come, attraverso il JSON, vengono presi i parametri necessari per impostare lo stato iniziale della simulazione. Vengono recuperate le informazioni relative al paziente, quali:

- Età
- Sesso
- Peso
- Stile di vita
- Insulina
- Glicemia
- Glicogeno
- Glucagone

Come si evince dal listato, il parametro JSON contiene anche una lista, i cui elementi appartengono alla classe *Food*. La lista rappresenta il dataset degli alimenti su cui sono disponibili alcune informazioni, quali: indice glicemico, grammi di proteine e grammi di carboidrati contenuti in 100g. Tali informazioni verranno utilizzate nel corso della simulazione per calcolare l'aumento della concentrazione di glucosio in seguito ai pasti.
Il metodo *start* definito dalla classe *Model*, invoca a sua volta il metodo *start* definito dalla classe *SimState*; il quale, come visibile dal codice, inizializza gli agenti che compongono il modello. Come si evince dal metodo *scheduleRepeating*, gli agenti verranno eseguiti dallo scheduler a ogni step. Inoltre, si inizializzano le istanze delle classi *Pda* e *Meals*. La prima classe rappresenta un'utility che consente di impostare, all'interno dell'ambiente di simulazione, il carico glicemico del pasto effettuato e i boli di insulina da simulare. La classe *Meals*, invece, contiene il dataset di alimenti con i relativi valori nutrizionali e, consente di calcolare per ciascun alimento, sulla base della quantità ingerita, le seguenti informazioni:

- Carico glicemico
- Grammi di proteine

- Grammi di carboidrati

Attraverso i metodi *setData(JSONObject json)* e *getSimulationData()*, il modello è in grado scambiare delle informazioni nel corso della simulazione. Al fine di offrire la massima flessibilità nel numero e nel formato dei dati da scambiare, i metodi accettano e restituiscono dei JSON.
Di seguito è presente l'implementazione dei metodi per il nostro caso di studio.

```
@Override
public void setData(JSONObject json) {
  try {
    if(json.has("meal")) {
      this.foodsInfo=(ArrayList<FoodInfo>)json.get("meal");
     InfoMeal meal=meals.getInfoMeal(this.foodsInfo);
     pda.insertMeal(sim, meal.getGlycmicLoad(),
         meal.getGramsCarbohydrates());
     _setProtein(meal.getProtein());
    }

    if(json.has("heartrate")) {
      int heartrate;
      if((heartrate=json.getInt("heartrate"))!=0) {
        if (isMale) {
          calories = (-55.0969 + (0.6309 * heartrate) + (0.1988 *
              getWeight()) + (0.2017 * getAge())) / (4.184 * 60 * 0.0167);
        } else {
          calories = (-20.4022 + (0.4472 * heartrate) - (0.1263 *
              getWeight()) + (0.074 * getAge())) / (4.184 * 60 * 0.0167);
        }
        _setActivity(new Activity("", 1, calories));
      }else{
        calories=0;
      }
    }else if(json.has("activity")) {
      _setActivity((Activity)json.get("activity"));
      calories=_getActivity().getCal();
    }

  } catch (JSONException e) {
    e.printStackTrace();
  }
}

public JSONObject getSimulationData() {
  JSONObject json=new JSONObject();
  try {
    json.put("step", schedule.getSteps());
    json.put("glycemia",getGlycemia());
    json.put("insulin",_getInsulin());
    json.put("glucagon", _getGlucagon());
```

```
40      json.put("liver_glycogen",liver_glycogen);
41      json.put("muscle_glycogen",muscle_glycogen);
42      if(_getActivity()!=null) {
43        json.put("calories", calories);
44
45          if(!_getActivityCheck()){
46            _setActivity(null);
47          }
48        }else{
49          json.put("calories",0);
50        }
51       if(boluses!=0){
52        json.put("boluses",boluses);
53        boluses=0;
54        }
55    }catch(JSONException e) {
56     e.printStackTrace();
57    }
58
59    return json;
60  }
```

Come già detto, il metodo *setData* consente di ricevere alcuni valori durante l'esecuzione degli step della simulazione. Come visibile dal codice, se nel JSON è presente il campo *meal*, il modello dovrà simulare un pasto. Pertanto, si prende l'elenco degli alimenti che costituiscono il pasto e, attraverso la classe di utility *Meal*, si ottengono le informazioni relative al pasto. Sulla base di queste informazioni, attraverso la classe di utility *Pda*, si calcolano i boli di insulina da simulare e l'aumento della concentrazione di glucosio nel sangue.
In caso di presenza del campo *heartrate*, è in corso una sessione real time. Tale campo contiene la frequenza cardiaca media del paziente e, attraverso questo valore, si stimano le calorie bruciate e si imposta l'attività fisica da simulare all'interno del modello.
In caso di presenza del campo *activity*, invece, si è nel caso di una simulazione settimanale. In tal caso si recupera l'attività fisica che si dovrà simulare per un numero di step definiti dall'oggetto.
Il metodo *getSimulationData* consente di ottenere dal simulatore i dati prodotti al termine dello step della simulazione. Come visibile, il JSON viene costruito passando i parametri relativi a:

- Glicemia
- Insulina
- Glucagone

- Glicogeno
- Calorie bruciate
- Boli di insulina da iniettare
- Step corrente della simulazione

Attraverso l'estensione della classe *Model*, si definiscono i metodi per leggere e inviare dei dati, lasciando allo sviluppatore la responsabilità di gestire eventuali concorrenze e trovare una soluzione per garantire lo scambio di dati a ogni step della simulazione.
Per agevolare lo sviluppo, dunque, la classe *Simulation* si occupa di gestire l'esecuzione del modello e lo scambio delle informazioni.
Il costruttore accetta come parametro un oggetto che estende la classe *Model*.
La classe espone i metodi *run(), pause(), stop(), resume()* che, come intuibile, consentono di: avviare, sospendere, interrompere e riprendere la simulazione.
Il metodo *run(JSON, int, int)* accetta come parametri:

- L'oggetto JSON che rappresenta i parametri da passare al modello prima di avviare la simulazione
- Il tempo di esecuzione tra uno step e quello successivo
- Il numero di step da simulare

Qualora sia il secondo che il terzo parametro dovessero essere uguali a -1, il modello verrà eseguito, senza effettuare pause tra uno step e quello successivo, fino a quando non verrà invocato il metodo *stop()* o *pause()*.
Per non bloccare il flusso di controllo, si affida l'esecuzione della simulazione a un thread; il quale esegue il modello fino a quando non si verifica una delle seguenti condizioni:

- Viene invocato il metodo *pause()* o il metodo *stop*
- Il modello termina l'esecuzione degli step

Il thread, nel corso della simulazione, invoca il metodo *nextStep()*. Tale metodo effettua le seguenti operazioni nell'ordine in cui vengono presentate:

- Effettua la callback su tutti gli observer interessati a inviare dei dati al modello per lo step successivo

- Esegue uno step della simulazione
- Effettua la callback su tutti gli observer interessati a ricevere i dati dal modello relativi allo step appena concluso

Oltre a poter definire degli observer per inviare e ricevere informazioni durante una simulazione, è possibile definire degli observer per monitorare lo stato della simulazione.

6.3 Invio richieste HTTP

Per l'interazione tra l'applicazione Android e il server, si è scelto di utilizzare l'API *Volley* presentata nei paragrafi precedenti.
Per effettuare una richiesta HTTP è sufficiente scrivere le seguenti righe di codice:

```
1  RequestQueue queue = Volley.newRequestQueue(Context);
2  StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
3     new Response.Listener<String>() {
4        @Override
5        public void onResponse(String response) {
6           ...
7        }
8     }, new Response.ErrorListener() {
9        @Override
10       public void onErrorResponse(VolleyError error) {
11          ...
12       }
13 });
14
15 queue.add(stringRequest);
```

Il metodo *newRequestQueue* della classe *Volley* consente di ottenere la coda attraverso la quale vengono inviate le richieste HTTP.
Successivamente si definisce la richiesta in base al tipo di dato che si vuole gestire nella risposta; nel caso del listato, una stringa. Il primo parametro del costruttore indica il metodo della richiesta HTPP (GET, POST, PUT, DELETE, etc). Il secondo parametro indica l'url a cui inviare la richiesta, mentre il terzo e l'ultimo parametro rappresentano le callback che verranno chiamate in caso di esito positivo o negativo della richiesta. Il parametro della callback, in caso di esito positivo, rappresenta il payload della risposta.
Volley fornisce inoltre il supporto per gestire le richieste tramite JSON. È sufficiente definire una richiesta del tipo:

```
1      JSONObject parameters= new JSONObject();
2
3      ...
4
5    JsonObjectRequest jsonObjectRequest = new JsonObjectRequest
6       (Request.Method.GET, url, parameters, new
            Response.Listener<JSONObject>() {
7          @Override
8          public void onResponse(JSONObject response) {
9             ...
10         }
11      }, new Response.ErrorListener() {
12         @Override
13         public void onErrorResponse(VolleyError error) {
14            ...
15         }
16      });
```

A differenza della precedente richiesta, in quest'ultima, i payload sono rappresentati in formato JSON.
Volley consente di estendere la classe *Request* e, attraverso di essa, è possibile creare delle classi che consentono di gestire il payload delle risposte delle richieste HTTP in diversi formati. Poiché l'applicazione dovrà inviare al server una richiesta per il download del programma Prolog associato al paziente, è necessario gestire un payload composto da dei byte. Di seguito è presente l'implementazione della classe *ByteRequest*; la quale consente di gestire delle richieste HTTP in cui il payload è costituito da un array di byte.

```
1  public class ByteRequest extends Request<byte[]> {
2    private final Response.Listener<byte[]> mListener;
3    private Map<String, String> mParams;
4    public Map<String, String> responseHeaders ;
5
6    public ByteRequest(int method, String mUrl ,Response.Listener<byte[]>
         listener, Response.ErrorListener errorListener, HashMap<String,
         String> params) {
7      super(method, mUrl, errorListener);
8      setShouldCache(false);
9      mListener = listener;
10     mParams=params;
11   }
12
13   @Override
14   protected void deliverResponse(byte[] response) {
15     mListener.onResponse(response);
16   }
17
18   @Override
```

```
19    protected Response<byte[]> parseNetworkResponse(NetworkResponse response)
          {
20      responseHeaders = response.headers;
21      return Response.success(response.data,
            HttpHeaderParser.parseCacheHeaders(response));
22    }
23  }
```

Per gestire una richiesta in cui il payload viene interpretato come un array di byte, è sufficiente estendere la classe *Request*, specificando il tipo di dato che si vuole parsare e, implementare i metodi astratti *deliverResponse(byte[] response)* e *parseNetworkResponse(NetworkResponse response)*.
Il metodo *parseNetworkResponse(NetworkResponse response)* ha come parametro un oggetto della classe *NetworkResponse*, il quale contiene: il payload della risposta, l'header e lo status code. Questo metodo è necessario per effettuare il parsing dei dati provenienti dalla risposta e restituire un array di byte.
Il metodo *deliverResponse(byte[] response)* viene invocato quando il metodo *parseNetworkResponse(NetworkResponse response)* restituisce una response. Ogni qualvolta che viene effettuata la callback sul metodo *deliverResponse(byte[] response)*, si effettua la callback sul listener; al fine di comunicare il payload della risposta alla richiesta HTTP inviata precedentemente.

6.4 Comunicazione Bluetooth

Similmente a quanto fatto per il body gateway, si vuole realizzare una libreria che consenta a uno sviluppatore, attraverso la tecnologia Bluetooth, di gestire la comunicazione con il device con il quale si è connessi.
Per Android non è necessario realizzare una classe in grado di accettare una connessione Bluetooth; in quanto è il device stesso a inviare la richiesta. Inoltre, Android fornisce delle API che consentono di effettuare il discovery di device Bluetooth. Pertanto, la libreria che si intende realizzare, si deve limitare a fornire gli strumenti necessari per gestire la comunicazione con il device con cui si intende stabilire la connessione.

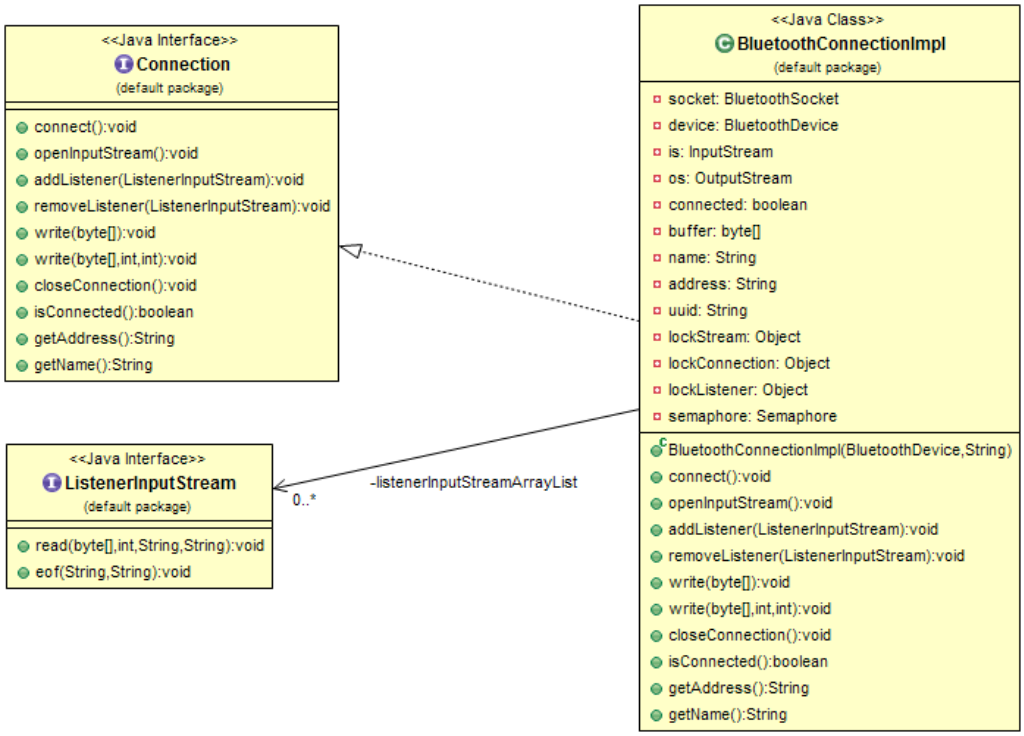

Figura 6.8: Diagramma delle classi per la comunicazione via Bluetooth

Come visibile dalla figura 6.8, le classi risultano essere simili a quelle presentate nel capitolo relativo al body gateway.
La classe *BluetoothConnectionImpl* fornisce le medesime funzionalità della sua omonima realizzata per il body gateway; si possono rilevare solo delle differenze a livello implementativo, legate alle API native fornite da Android.

6.5 Invio e ricezione dati

Il body gateway è in grado di ricevere e rispondere alle richieste di acquisizione dati. In modo complementare, l'applicazione Android deve essere in grado di inviare le richieste di acquisizione dati e ricevere le relative risposte.
Sulla base dell'API realizzata per il body gateway, l'API per Android può essere concepita come illustrato nel diagramma delle classi presente nella figura successiva.

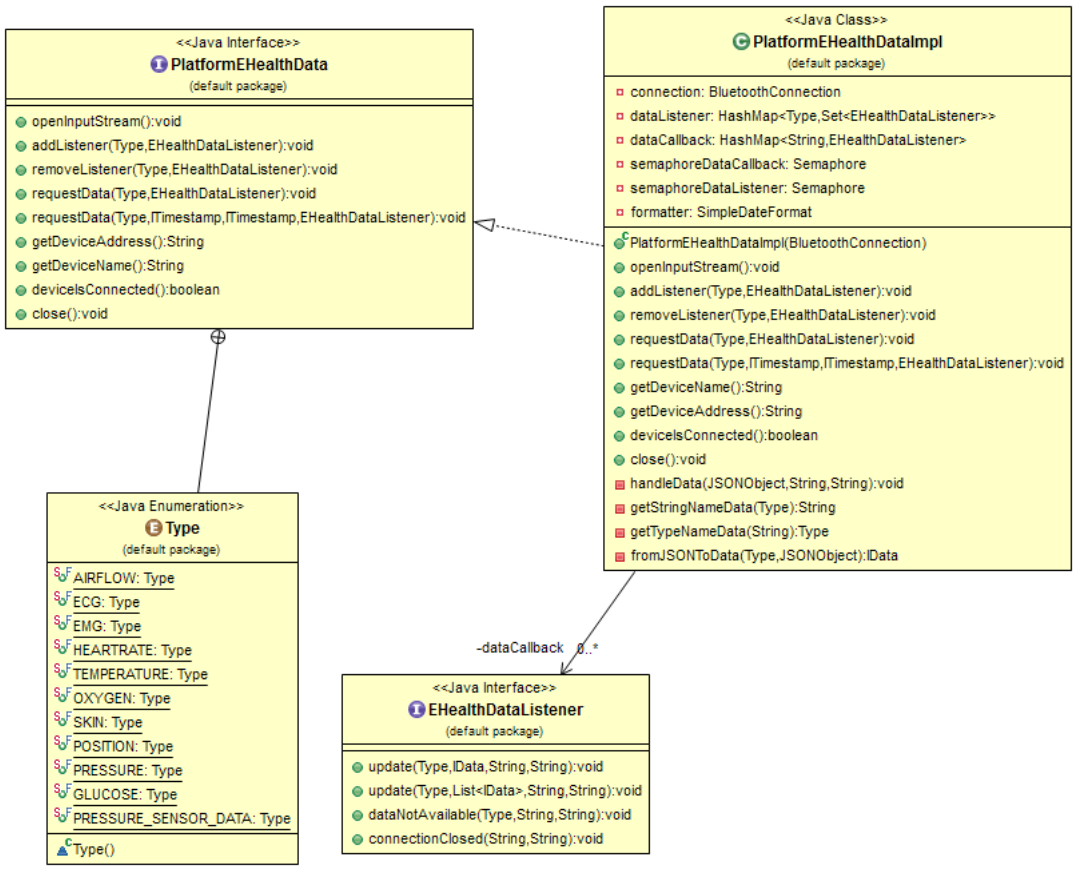

Figura 6.9: Diagramma delle classi per lo scambio dei dati

La classe *PlatformEHealthDataImpl* fornisce i metodi necessari per inviare le richieste di acquisizione dati e per gestire la loro ricezione.
Come si evince dal diagramma, per consentire di ricevere dei dati senza un'esplicita richiesta, è necessario implementare il pattern *Observer*. Attraverso il metodo *addListener*, è possibile aggiungere un listener per un certo tipo di dato.
Attraverso i metodi *requestData* è invece possibile richiedere i dati attraverso una richiesta esplicita. Come si evince dal diagramma, è possibile richiedere il dato con o senza l'intervallo di tempo in cui i dati sono stati generati.
Il costruttore della classe accetta come parametro l'oggetto che implementa la classe *Connection*. In questo modo è possibile inviare le richieste di acquisizione dati e ricevere quest'ultimi attraverso diverse tecnologie di comunicazione;

non obbligando lo sviluppatore a utilizzare la tecnologia Bluetooth.
Al momento della creazione dell'oggetto, si aggiunge un observer sui dati provenienti dal device. Ogni qualvolta che viene invocato il metodo *read*, i byte vengono interpretati come char. Fino a quando non viene letto il carattere di nuova riga ('\n'), i caratteri letti fino a quel momento vengono concatenati. Al momento della ricezione del carrattere di nuova riga, la stringa formatosi fino a quel momento viene interpretata secondo la sintassi JSON. Se la stringa è compatibile con la sintassi JSON, si tenta di recuperare il tipo di dato ricevuto tramite il metodo *handleData*. Tale metodo si occupa di verificare la presenza o meno del dato, controllando la presenza del campo *notAvailable* all'interno del JSON. Successivamente, verifica il tipo di dato a cui è associata l'informazione, in modo da definire su quali observer effettuare la procedure call per notificare l'arrivo di un nuovo dato da parte del body gateway. Infine, verifica la presenza, all'interno del JSON, del campo *requestCode*. Se esso è presente, allora il dato è stato inviato dal body gateway in seguito a una richiesta di acquisizione; altrimenti è stato inviato senza una richiesta esplicita di acquisizione dati. Il metodo, in base a queste informazioni, effettua una procedure call sugli observer associati al tipo di dato ricevuto e, se presente il campo *requestCode* all'interno del JSON, effettua una procedure call sulla callback associata per quel particolare identificativo.
Attraverso i metodi *requestData* è possibile inviare una richiesta di acquisizione dati. Poiché non è possibile stimare se e quando si riceverà una risposta, è necessario definire un observer sulla quale verrà effettuata la callback al momento della ricezione del dato. Trattandosi di richieste asincrone, se ne potrebbero effettuare più di una prima che il gateway invii le risposte. Pertanto, per identificare l'observer sul quale effettuare la callback in modo corretto, al momento dell'invio della richiesta, si associa un identificativo; il quale è dato da un timestamp che rappresenta l'istante di tempo in cui la richiesta viene inviata.

6.6 Pairing tra device Bluetooth

Per effettuare il discovery dei device Bluetooth, sono necessarie le seguenti righe di codice:

```
1  IntentFilter filter = new IntentFilter();
2  filter.addAction(BluetoothDevice.ACTION_FOUND);
3  registerReceiver(receiver, filter);
```

Come visibile, si registra un receiver; un componente che ci permette di essere notificati quando si verificano degli eventi del sistema o dell'applicazione. Gli eventi a cui siamo interessati vengono definiti attraverso un oggetto della classe *IntentFilter*. Nel nostro caso, siamo interessati all'evento in cui un dispositivo Bluetooth viene individuato. Al verificarsi di tale evento, verrà effettuata una callback sul listener passato al metodo *registerReceiver*. Nel nostro caso, l'azione sarà la seguente:

```
1  BroadcastReceiver receiver = new BroadcastReceiver() {
2       public void onReceive(Context context, Intent intent) {
3          String action = intent.getAction();
4          if (BluetoothDevice.ACTION_FOUND.equals(action)) {
5             BluetoothDevice device =
                  intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
6             BluetoothConnection bluetoothConnection=new
                  BluetoothConnectionImpl(device,
                  "04c6093b-0000-1000-8000-00805f9b34fb");
7             bluetoothConnection.connect();
8             BluetoothEHealthData platform=new
                  BluetoothEHealthDataImpl(bluetoothConnection);
9             platform.openInputStream();
10            EHealthPlatformMemory.setPlatform(platform);
11
12         }
13      }
14 };
```

Quando si verifica un evento di tipo *ACTION_FOUND*, un dispositivo Bluetooth è stato individuato. Attraverso il metodo *getParcelableExtra*, è possibile ottenere un oggetto POJO che rappresenta il device Bluetooth individuato in fase di discovery.
L'oggetto della classe *BluetoothDevice* rappresenta il device con il quale vogliamo stabilire una connessione Bluetooth; nel nostro caso il body gateway. Stabilita la connessione, si crea l'oggetto che consente di ottenere i dati biomedici provenienti dal body gateway. Una volta aperto lo stream di input, al fine di leggere i dati ricevuti, si aggiunge l'oggetto appena creato all'interno di una classe statica; il cui ruolo è quello di mantenere un'istanza della classe in qualsiasi activity dell'applicazione Android.
La connessione con il body gateway avviene attraverso l'utilizzo delle API native di Android. Il metodo *connect* della classe *BluetoothConnection*, per effettuare il pairing, esegue le istruzioni presenti nel listato sottostante.

```
1   public void connect(){
2      BluetoothSocket socket=device.createRfcommSocketToServiceRecord(UUID);
3      socket.connect();
4   }
```

Attraverso il metodo *createRfcommSocketToServiceRecord(UUID)* si inizializza una socket Bluetooth, attraverso la quale è possibile connettersi con il device desiderato. L'uuid deve coincidere con quello usato dal device che agisce come server.
Tramite il metodo *connect* fornito dalla socket, è possibile stabilire la connessione. All'invocazione del metodo, il sistema esegue un SDP lookup per trovare il device remoto con l'uuid indicato in precedenza. Nel caso in cui si riesca a stabilire una connessione, verrà condiviso un canale RFCOMM; attraverso il quale sarà possibile comunicare.

6.7 Sincronizzazione misurazioni

Una volta stabilita la connessione con il body gateway, il paziente è in grado di acquisire le misurazioni presenti sulla memoria del glucometro e dello sfigmomanometro. Si è deciso di implementare una forma di sincronizzazione tra il server e il body gateway; al fine di consentire al paziente di visualizzare le misurazioni, indipendente dal device che utilizza. Di seguito è possibile visualizzare il listato di codice che consente di inviare una richiesta di acquisizione dati, attraverso l'ausilio delle API illustrate poc'anzi.

```
1  ...
2
3  public void syncData(){
4    BluetoothEHealthData platform=EHealthPlatformMemory.getPlatform();
5    if (platform != null && platform.deviceIsConnected()) {
6      try {
7        Timestamp to=new Timestamp(Calendar.getInstance());
8        Calendar tmp=null;
9        synchronized (lock) {
10         Timestamp from;
11         synchronized (lockFile) {
12           from = model.getLastSynchronization(activity);
13         }
14         if (activity.equals("pressure")) {
15           platform.requestData(BluetoothEHealthData.
                 Type.PRESSURE_SENSOR_DATA,from,to,listener);
16         } else {
17           platform.requestData(BluetoothEHealthData.Type.GLUCOSE,from,to,
                 listener);
```

```
18          }
19        }
20      } catch (Exception e) {
21       e.printStackTrace();
22      }
23    }else{
24      getDataFromServer();
25      view.deviceNotConnected();
26    }
27  }
28
29  private void getDataFromServer(){
30
31    synchronized (lock) {
32      HttpRequest.getInstance(context).sendStringRequest(Request.Method.GET,
            ipaddress+"/get-" + (activity.equals("pressure") ? "pressure" :
            "glycemia") + "?username=" + UserInfo.getUsername(), new
            Response.Listener<String>() {
33        @Override
34        public void onResponse(String response) {
35          try {
36            JSONObject responseJson = new JSONObject(response);
37            synchronized (lock) {
38              synchronized (lockFile) {
39                model.writeData(responseJson.getJSONArray("data"), activity);
40              }
41            }
42          } catch (JSONException e) {
43            e.printStackTrace();
44          }
45        }
46      }, new Response.ErrorListener() {
47        @Override
48        public void onErrorResponse(VolleyError error) {
49          error.printStackTrace();
50        }
51      });
52    }
53  }
54
55  ...
```

Come visibile dal listato, nel metodo *syncData*, si recupera l'oggetto che consente di inviare le richieste per l'acquisizione dei dati biomedici.
Qualora il device Android non dovesse essere connesso via Bluetooth con il body gateway, verrà mostrato all'utente un messaggio che segnala tale problema e verrà inviata al server una richiesta per ottenere tutte le misurazioni effettuate dal paziente. Quest'ultime verranno memorizzate nello storage interno del device, al fine di permettere una loro visualizzazione anche in assenza di connessione Internet.

In caso di connessione Bluetooth con il body gateway, sulla base del tipo di acquisizione dati richiesta del paziente, si procede con l'invio della richiesta. Per evitare di ottenere tutte le misurazioni già presenti sul server, si richiede al body gateway le misurazioni effettuate tra la data corrente e la data relativa all'ultima sincronizzazione con il server. Al momento della ricezione dei dati inviati dal body gateway, viene effettuata la callback sul listener.

```
1  EHealthDataListener listener=new EHealthDataListener() {
2
3    @Override
4    public void update(BluetoothEHealthData.Type type, IData value, String
         name, String address) {
5    }
6
7    @Override
8    public void update(final BluetoothEHealthData.Type type, final
         List<IData> values, String name, String address) {
9      JSONObject json=new JSONObject();
10     String value="";
11     try {
12       json.put("username", UserInfo.getUsername());
13       synchronized (lock) {
14         json.put("data", model.writeData(values,activity));
15         value=activity.equals("pressure")?"pressure":"glycemia";
16       }
17     } catch (JSONException e) {
18       e.printStackTrace();
19     }
20
21     HttpRequest.getInstance(context).sendJSONRequest(Request.Method.POST,
           ipaddress+"/add-"+value, json, new Response.Listener<JSONObject>() {
22       @Override
23       public void onResponse(JSONObject response) {
24         getDataFromServer();
25       }
26     }, new Response.ErrorListener() {
27       @Override
28       public void onErrorResponse(VolleyError error) {
29         error.printStackTrace();
30       }
31     });
32   }
33
34   @Override
35   public void dataNotAvailable(final BluetoothEHealthData.Type type, String
         name, String address) {
36     getDataFromServer();
37     view.dataNotAvailable();
38   }
39
40   @Override
```

```
41   public void connectionClosed(String name, String address) {}
42 };
```

L'invocazione del metodo *dataNotAvailable* denota l'assenza di misurazioni nell'intervallo di tempo richiesto. In tal caso si mostra al paziente un messaggio che indica l'assenza di nuove misurazioni sul device medico, e si invia al server una richiesta per ottenere lo storico delle misurazioni effettuate dal paziente; in modo da aggiornare il file salvato in locale.
L'invocazione del metodo *update* denota la presenza di misurazioni nell'intervallo di tempo richiesto. In tal caso si salvano le nuove misurazioni in locale. Il metodo utilizzato per il salvataggio in locale delle misurazioni restituisce quest'ultime in formato JSON; le quali vengono inserite in un JSONObject che viene inviato tramite una richiesta al server; il quale memorizza le nuove misurazioni acquisite dal device medico.

6.8 Elaborazione feedback

In fase di autenticazione, l'applicazione Android invia una richiesta HTTP GET al server; per scaricare, se presente, il file di regole associato al paziente. Pertanto, l'operazione da effettuare al momento del login, è la seguente.

```
1  public void login(final String username, final String password){
2    JSONObject json=new JSONObject();
3
4    try {
5      json.put("username",username);
6      json.put("password",password);
7    } catch (JSONException e) {
8      e.printStackTrace();
9    }
10
11   HttpRequest.getInstance(context).sendJSONRequest( Request.Method.POST,
         ipaddress+"/login",json,new Response.Listener<JSONObject>() {
12     @Override
13     public void onResponse(JSONObject response) {
14       ...
15
16       HttpRequest.getInstance(context).sendByteRequest(Request.Method.GET,
             ipaddress+"/get-rule-file?patient=" + username, new
             HashMap<String, String>(), new Response.Listener<byte[]>() {
17         @Override
18         public void onResponse(byte[] response) {
19           if (response != null) {
20             FileOutputStream outputStream = context.openFileOutput(
                   "diabetes.pl",           Context.MODE_PRIVATE);
```

```
            outputStream.write(response);
            outputStream.close();
          }
        }
      }, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
          error.printStackTrace();
        }
      });
    }
  }, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {
      error.printStackTrace();
    }
  });
}
```

Come visibile dal listato, quando l'autenticazione avviene con successo, si invia una richiesta per l'acquisizione del file e, se il payload della risposta non è vuoto, si crea un file Prolog in cui il contenuto corrisponde al payload della risposta proveniente dal server.
Naturalmente Android non supporta l'esecuzione di file Prolog e, per utilizzare tale linguaggio, è necessario l'interprete *JIProlog*.
L'interprete Prolog è definito dalla classe *JIPEngine*. Di seguito è possibile notare l'implementazione della classe che si occupa di restituire il feedback elaborato dall'interprete Prolog.

```
public class PrologInterpreter {
    private static PrologInterpreter interpreter;
    private JIPEngine jip;

    private PrologInterpreter(){
        interpreter=new PrologInterpreter();
        jip=new JIPEngine();
    }

    public static PrologInterpreter getInstance(){
        if(interpreter==null){
            interpreter=new PrologInterpreter();
        }

        return interpreter;
    }

    public JSONArray getFeedback(InputStream stream, JSONArray parameters,
        JSONArray constraints){
        jip.consultStream(stream, "stream");
```

```
20
21          JIPFunctor query = (JIPFunctor)
                jip.getTermParser().parseTerm("function_call( checkValue, " +
                parameters.toString() + "," + constraints.toString() +
                ",[(-1,1,3), (1,1,3), (-1,1,3), (1,1,3)], X).");
22
23          JIPQuery jipQuery = jip.openSynchronousQuery(query);
24          JIPTerm solution = null;
25          solution = jipQuery.nextSolution();
26
27          JIPList list = (JIPList) solution.getVariables()[0].getValue();
28          JSONArray week = new JSONArray();
29
30          for (int i = 1; i < list.length(); i++) {
31              JIPList day = (JIPList) list.getNth(i).getValue();
32              JSONArray tipsDay = new JSONArray();
33              System.out.println("Giorno: " + i);
34              for (int j = 1; j <= day.length(); j++) {
35
36                  JIPList tips = (JIPList) day.getNth(j).getValue();
37                  System.out.println("Consiglio: " + j + " " +
                        tips.getNth(1).getValue().toString() + " " +
                        tips.getNth(2).getValue().toString());
38                  JSONObject tip = new JSONObject();
39                  tip.put("meal", tips.getNth(1).getValue().toString());
40                  tip.put("activity", tips.getNth(2).getValue().toString());
41                  tipsDay.put(tip);
42              }
43              week.put(tipsDay);
44          }
45
46          return week;
47      }
48  }
```

Il metodo *getFeedback()* permette di ottenere il feedback elaborato dall'interprete Prolog. Accetta in input il file Prolog sotto forma di *InputStream*, l'elenco dei parametri e l'elenco dei vincoli necessari per ottenere il feedback. All'invocazione del metodo, tramite la funzione *consultStrream* della classe *JIPEngine*, si definisce il file da caricare e, in seguito, si costruisce, tramite il parsing di una stringa, il termine. Definito il termine, si effettua la query e, attraverso il metodo *solution.getVariables()*, si recupera il risultato unificato con la variabile *X*.
Come illustrato nel quinto capitolo, la soluzione prodotta dall'insieme di regole, restituisce una lista in cui ogni elemento rappresenta un giorno e, a sua volta, un giorno è costituito da una lista in cui ogni elemento rappresenta un consiglio costituito da un termine di due elementi (regime alimentare e atti-

vità fisica consigliate). Pertanto, si itera la lista al fine di restituire le possibili soluzioni sotto forma di array JSON.

6.9 Simulazione settimanale

La simulazione settimanale ha lo scopo di fornire una previsione dell'andamento glicemico del paziente nell'arco della settimana. Il modello fa affidamento sui pasti e sull'attività fisica che l'utente intende effettuare nell'arco della settimana simulata. Tali informazioni vengono fornite dal paziente attraverso il livello di presentazione dell'applicazione.
All'avvio della simulazione, il controller effettua alcune operazioni. La prima, in ordine cronologico, è quella relativa alla preparazione del modello e del relativo passaggio dei dati forniti dal paziente.

```
1  public void setActivities(List<PhysicalActivity> activities){
2
3    this.activities.clear();
4
5    for(int i=0;i<caloriesActivitiesWeek.length;i++){
6      caloriesActivitiesWeek[i]=0;
7    }
8
9    for(PhysicalActivity activity:activities){
10     int step=getStep(activity.getDay(),activity.getStartTime());
11     this.activities.put(step,new
           Activity("",activity.getPeriod(),activity.getCalories()));
12     int index=getDay(step);
13     caloriesActivitiesWeek[index]+=activity.getCalories()*
           activity.getPeriod();
14   }
15 }
```

Attraverso il metodo *setActivities*, si memorizzano in quali step della simulazione il modello dovrà tenere conto dell'attività fisica che svolgerà il paziente durante la settimana. Inoltre, per ogni giorno della settimana, si effettua il calcolo delle calorie bruciate dal paziente.

```
1  public void setMeals(List<Meal> meals){
2
3    this.meals.clear();
4    glycemiaMeal.clear();
5
6    for(int i=0;i<glycemicLoadMealsWeek.length;i++){
7      glycemicLoadMealsWeek[i]=0;
8      glycemicLoadMealsWeekCount[i]=0;
```

```
  }

  for(Meal meal:meals){
    int step=getStep(meal.getDay(),meal.getTime());

    if (this.meals.indexOfKey(step)>=0) {
      this.meals.get(step).addAll(meal.getFoods());
    } else {
      this.meals.put(step, meal.getFoods());
      glycemiaMeal.add(new GlycemiaMealInfo(step,new Pair<>(0L,0.0),new
          Pair<>(0L,0.0),new Pair<>(10080L,Double.MAX_VALUE)));
      int index=getDay(step);
      glycemicLoadMealsWeek[index]+=meal.getGlycemicLoadMeal();
      glycemicLoadMealsWeekCount[index]+=meal.getFoods().size();
    }
  }
}
```

Attraverso il metodo *setMeals*, invece, si memorizzano in quali step della simulazione il modello dovrà tenere conto dei pasti che il paziente effettuerà. Inoltre, per ogni giorno della settimana, si memorizzano il numero dei pasti che effettuerà l'assistito e il carico glicemico complessivo.
Per avviare la simulazione, è sufficiente invocare il metodo sottostante.

```
public void run(JSONObject params){
  simulation.run(params,-1,10080);
}
```

Come già visto nei paragrafi precedenti, il metodo *run* della classe *Simulation*, consente di avviare la simulazione passando i parametri necessari al modello; quest'ultimi consentono di definire lo stato iniziale del modello. I parametri che può definire il paziente, attraverso l'interfaccia grafica sono:

- Peso
- Glicemia
- Insulina
- Glucagone
- Glicogeno fegato
- Glicogeno muscoli

Qualora l'utente non dovesse inserire uno o più parametri, verranno impostati dei valori di default.
Come si evince dai restanti parametri, la simulazione verrà effettuata per una durata di 10080 step. Tale numero rappresenta il numero di minuti presenti in una settimana. Inoltre, non verranno effettuate interruzioni tra uno step e quello successivo.
Poiché siamo interessati a inviare e ricevere dati dal modello, nel corso della simulazione è necessario aggiungere un observer. Di seguito, l'implementazione dell'observer per il nostro caso di studio.

```
simulation.addObserverRequestData(new ObserverRequestData() {
  @Override
  public JSONObject requestData() {
    int step=(int)simulation.getSteps();
    if (meals.indexOfKey(step)>=0 || activities.indexOfKey(step)>=0) {
      JSONObject jsonObject = new JSONObject();
      try {
        if (meals.indexOfKey(step)>=0) {
        jsonObject.put("meal", meals.get(step));
        }

        if (activities.indexOfKey(step)>=0) {
          jsonObject.put("activity", activities.get(step));
        }

        return jsonObject;
      } catch (JSONException e) {
        e.printStackTrace();
      }
   }
   return null;
  }
});
```

Ad ogni step, all'invocazione del metodo *requestData*, si controlla che ci sia un pasto o un'attività fisica. Qualora fosse previsto un pasto o un'attività, si imposta il JSON e lo si restituisce attraverso il metodo. Per ricevere i dati nel corso della simulazione, tra uno step e quello successivo, invece, basta definire l'observer illustrato nella sezione relativa alle modifiche del modello. Di seguito, l'implementazione dell'observer per il nostro caso di studio.

```
simulation.addObserverData(new ObserverDataSimulation() {
  @Override
  public void update(JSONObject data) {
    try{
      long step=simulation.getSteps();

```

```
7       System.out.println(step);
8
9       double glycemia=data.getDouble("glycemia");
10      double insulin=data.getDouble("insulin");
11      double glucagon=data.getDouble("glucagon");
12      double calories=data.getDouble("calories");
13
14      for(GlycemiaMealInfo tmp:glycemiaMeal){
15        if(step==tmp.getStep()){
16          tmp.setGlycemiaPrePrandial(new
                Pair<>(simulation.getSteps(),glycemia));
17        }
18
19        if(step>=tmp.getStep() && step<=(tmp.getStep()+120)){
20          if(tmp.getGlycemiaMax().second<glycemia){
21            tmp.setGlycemiaMax(new Pair<>(simulation.getSteps(),glycemia));
22          }
23        }
24
25        if(step>=(tmp.getStep()+120) &&step<=(tmp.getStep()+240)){
26          if(tmp.getGlycemiaPostPrandial().second>glycemia){
27            tmp.setGlycemiaPostPrandial(new
                  Pair<>(simulation.getSteps(),glycemia));
28          }
29        }
30      }
31
32      JSONObject tmp=new JSONObject();
33      tmp.put("step",step);
34      tmp.put("glycemia",glycemia);
35      tmp.put("insulin",insulin);
36      tmp.put("glucagon",glucagon);
37      tmp.put("calories",calories);
38
39      JSONArray arrayMeal=new JSONArray();
40
41      if (meals.indexOfKey((int)step)>=0) {
42        List<FoodInfo> list=meals.get((int)step);
43        for(FoodInfo food:list){
44          arrayMeal.put(food.toJSON());
45        }
46      }
47
48      tmp.put("meal",arrayMeal);
49      jsonArray.put(tmp);
50
51      view.newData(glycemia,glucagon,insulin,(int)step);
52     }catch (JSONException e){
53       e.printStackTrace();
54     }
55    }
56  });
57 }
```

Dallo step corrente, attraverso il parametro JSON passato attraverso la callback, si recuperano le informazioni relative a: glicemia, glucagone, glicogeno e calorie bruciate nello step corrente.
Se nello step corrente è in corso un pasto, si memorizza il valore della glicemia pre-prandiale per il pasto in corso. Se è in corso il processo di digestione, invece, si prende il picco massimo della glicemia nell'arco delle due ore successive all'inizio del pasto. Se si è tra le due e le quattro ore successive al pasto, invece, si memorizza il valore minimo della glicemia. I valori glicemici acquisiti nelle diverse fasi del pasto, sono necessari per elaborare le informazioni da fornire alle regole Prolog; al fine di ottenere i suggerimenti sullo stile di vita da seguire. Le informazioni relative alla glicemia, al termine della simulazione, vengono processate, al fine di calcolare, per ogni giorno della settimana, i valori necessari al sistema di regole per produrre un feedback. I valori ricavati dai dati della simulazione sono:

- Picco glicemico medio post prandiale
- Coefficiente angolare medio tra i picchi glicemici post prandiali e il valore glicemico al momento del pasto
- Coefficiente angolare medio tra i picchi glicemici post prandiali e il minimo valore glicemico registrato tra le due e quattro ore successive al pasto

Le informazioni fornite dal modello nel corso della simulazione, inoltre, vengono memorizzate per l'invio al server una volta terminata la simulazione.
Come visto durante la presentazione delle API realizzate, è possibile osservare lo stato del simulatore attraverso l'observer presente nel listato che segue.

```
simulation.addObserverState(new ObserverState() {
  @Override
  public void stateChanged(int state) {
    ...
  }
});
```

Pertanto, al termine della simulazione, si effettuano le operazioni definite nel listato sottostante.

```
if(Simulation.COMPLETE==state){
       view.complete();

```

```
4         try {
5           double[] averageGlycemia = new double[7];
6           double[] alpha = new double[7];
7           double[] beta = new double[7];
8
9           int[] count = new int[7];
10
11          for (GlycemiaMealInfo mealInfo : glycemiaMeal) {
12            double tmpAlpha = (Math.abs((mealInfo.getGlycemiaMax().second -
                  mealInfo.getGlycemiaPrePrandial().second) /
                  (mealInfo.getGlycemiaMax().first -
                  mealInfo.getGlycemiaPrePrandial().first)));
13            double tmpBeta =
                  (Math.abs((mealInfo.getGlycemiaPostPrandial().second -
                  mealInfo.getGlycemiaMax().second) /
                  (mealInfo.getGlycemiaPostPrandial().first -
                  mealInfo.getGlycemiaMax().first)));
14
15            if (tmpAlpha == NaN) {
16              tmpAlpha = 0;
17            } else {
18              tmpAlpha = Utility.round(tmpAlpha, 4);
19            }
20
21            if (tmpBeta == NaN) {
22              tmpBeta = 0;
23            } else {
24              tmpBeta = Utility.round(tmpBeta, 4);
25            }
26
27            int index = getDay(mealInfo.getStep());
28            averageGlycemia[index] += mealInfo.getGlycemiaMax().second;
29            alpha[index] += tmpAlpha;
30            beta[index] += tmpBeta;
31            count[index]++;
32          }
33
34          JSONArray parametersProlog = new JSONArray();
35          JSONArray constraintsProlog = new JSONArray();
36          String[] glycemicLoadArray = new String[7];
37          String[] activityArray = new String[7];
38
39          for (int i = 0; i < 7; i++) {
40            JSONArray parameter = new JSONArray();
41            JSONArray constraint = new JSONArray();
42            int glycemicLoadLevel = 1;
43            int caloriesLevel = 1;
44
45            if (glycemicLoadMealsWeekCount[i] != 0) {
46              double avgGlycemicLoad = glycemicLoadMealsWeek[i] /
                    glycemicLoadMealsWeekCount[i];
47              if (avgGlycemicLoad < 10) {
48                glycemicLoadLevel = 1;
```

```
49              glycemicLoadArray[i] = "low";
50            } else if (avgGlycemicLoad >= 20) {
51              glycemicLoadLevel = 3;
52              glycemicLoadArray[i] = "high";
53            } else {
54              glycemicLoadLevel = 2;
55              glycemicLoadArray[i] = "medium";
56            }
57          } else {
58            glycemicLoadLevel = 1;
59            glycemicLoadArray[i] = "low";
60          }
61
62          if (caloriesActivitiesWeek[i] < 2500) {
63            caloriesLevel = 1;
64            activityArray[i] = "low";
65          } else if (caloriesActivitiesWeek[i] >= 5000) {
66            caloriesLevel = 3;
67            activityArray[i] = "high";
68          } else {
69            caloriesLevel = 2;
70            activityArray[i] = "medium";
71          }
72
73          if (count[i] != 0) {
74            parameter.put(averageGlycemia[i] / count[i]);
75            parameter.put(alpha[i] / count[i]);
76            parameter.put(beta[i] / count[i]);
77          } else {
78            parameter.put(0);
79            parameter.put(0);
80            parameter.put(0);
81          }
82          parameter.put(glycemicLoadLevel);
83          parameter.put(caloriesLevel);
84          constraint.put(constraintFood[i].first);
85          constraint.put(constraintFood[i].second);
86          constraint.put(constraintActivity[i].first);
87          constraint.put(constraintActivity[i].second);
88
89          parametersProlog.put(parameter);
90          constraintsProlog.put(constraint);
91        }
92
93        JSONObject dataSimulation = new JSONObject();
94        dataSimulation.put("data", jsonArray);
95        dataSimulation.put("username",
              ModelControllerSimulationDiabetes.this.username);
96        dataSimulation.put("date", date);
97
98        HttpRequest.getInstance(context).sendJSONRequest(Request.Method.POST,
              ipaddress+"/send-data-simulation", dataSimulation, new
              Response.Listener<JSONObject>() {
```

```
99            @Override
100           public void onResponse(JSONObject response) {
101           }
102         }, new Response.ErrorListener() {
103           @Override
104           public void onErrorResponse(VolleyError error) {
105           }
106         });
107
108         File file = new File(context.getFilesDir(), "diabetes.pl");
109         if (file.exists()) {
110
111           JSONObject feedback=new JSONObject();
112           feedback.put("data",
                  PrologInterpreter.getInstance().getFeedback(
                  context.openFileInput( "diabetes.pl"), parametersProlog,
                  constraintsProlog));
113           view.feedback(feedback,glycemicLoadArray,activityArray);
114         } else {
115           view.feedback(new JSONObject(),glycemicLoadArray,activityArray);
116         }
117       }catch(JSONException | FileNotFoundException e){
118         e.printStackTrace();
119       }
120     }
```

I dati relativi a ogni pasto vengono utilizzati per calcolare la media dell'indice glicemico per ogni giorno della settimana e, per calcolare, la relativa fascia (basso, medio ed elevato).
Analogamente, viene effettuato un calcolo simile per l'attività fisica. Si effettua, per ogni giorno della settimana, la somma delle calorie bruciate e si associa un valore (basso, medio ed elevato).
Tali informazioni vengono fornite al sistema di regole definite dal medico curante, il quale produce un feedback che viene mostrato al paziente attraverso il livello di presentazione. Infine, le informazioni memorizzate nel corso della simulazione, vengono mostrate al paziente e inviate al server.

6.10 Sessione real time

Il controller di questa funzionalità è strutturato in modo analogo a quello della simulazione, con alcune differenze dal punto di vista implementativo.
Nella sessione real time, uno dei dati che viene passato al modello, nel corso della simulazione, è la frequenza cardiaca. Pertanto, è necessaria l'interazione con il body gateway.

Un'altra differenza che contraddistingue la sessione real time dalla simulazione settimanale, dal punto di vista implementativo, è legata all'invio dei dati prodotti al server. Poiché il medico potrebbe voler controllare la sessione in tempo reale, è necessario inviare al server le informazioni della simulazione minuto per minuto, e non al termine della sessione.
Infine, l'ultima differenza, è legata dal tempo di esecuzione tra uno step e quello successivo. Poiché la sessione simula l'andamento glicemico in tempo reale, minuto per minuto, tra uno step e quello successivo deve intercorrere un minuto.
Di seguito è presente il listato di codice che mostra l'avvio della sessione real time.

```
1  public void run(JSONObject params, BluetoothEHealthData device,double
       glycemia,double glucagon,double insulin,double liver_glycogen,double
       muscle_glycogen){
2    this.device=device;
3    date=new SimpleDateFormat("YYYY-MM-dd
       HH:mm:ss").format(Calendar.getInstance().getTime()).replace(" ","T");
4
5    super.run(params,60000,-1);
6
7    JSONObject params2=new JSONObject();
8
9    try {
10
11     params2.put("username",username);
12     params2.put("id_session",username+"_"+date);
13     params2.put("date",date);
14     params2.put("new",true);
15     params2.put("glycemia",glycemia);
16     params2.put("glucagon",glucagon);
17     params2.put("insulin",insulin);
18     params2.put("liver_glycogen",liver_glycogen);
19     params2.put("muscle_glycogen",muscle_glycogen);
20     params2.put("calories",0);
21     params2.put("meal",new JSONArray());
22   } catch (JSONException e) {
23     e.printStackTrace();
24   }
25
26   HttpRequest.getInstance(context).sendJSONRequest(Request.Method.POST,
       ipaddress+"/send-data", params2, new Response.Listener<JSONObject>() {
27     @Override
28     public void onResponse(JSONObject response) {
29     }
30   }, new Response.ErrorListener() {
31     @Override
32     public void onErrorResponse(VolleyError error) {
33       error.printStackTrace();
```

```
    }
  });
}
```

Come visibile dal listato, si avvia la simulazione con un intervallo di 1 minuto e per un tempo indefinito. All'avvio della simulazione, inoltre, si inviano i dati al server per comunicare l'inizio della simulazione.
Per quanto concerne lo stato della simulazione, quando quest'ultima viene avviata o interrotta, si effettuano le operazioni visibili nel listato sottostante.

```
if(state==Simulation.RUN){
  if(device!=null) {
    threadHeartRate=new Thread(){
      public void run(){
        synchronized (lock) {
          bpmSum = 0;
          count = 0;
        }
        boolean flag=true;
        flagThreadHeartRate=true;

        while(flag){
          try {
            device.requestData(BluetoothEHealthData.Type.HEARTRATE,listener);
          } catch (IOException e) {
            e.printStackTrace();
          }
          try {
            Thread.sleep(1000);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          synchronized (lockHeartRate){
            flag=flagThreadHeartRate;
          }
        }
      }
    };
    threadHeartRate.start();
  }

  view.run();
}else if(state==Simulation.STOP){
  if(device!=null) {
    synchronized (lockHeartRate){
      flagThreadHeartRate=false;
    }
  }

  JSONObject params=new JSONObject();
  SimpleDateFormat format=new SimpleDateFormat("YYYY-MM-dd HH:mm:ss");
```

```
42
43    try {
44      params.put("username",username);
45      params.put("id_session",username+"_"+date);
46      params.put("date",format.format(Calendar.getInstance().getTime()).
            replace(" ","T"));
47      params.put("end",true);
48    } catch (JSONException e) {
49      e.printStackTrace();
50    }
51
52    HttpRequest.getInstance(context).sendJSONRequest(Request.Method.POST,
          ipaddress+"/send-data", params, new Response.Listener<JSONObject>() {
53        @Override
54        public void onResponse(JSONObject response) {}
55      }, new Response.ErrorListener() {
56        @Override
57        public void onErrorResponse(VolleyError error) {}
58      });
59    ...
60  }
```

Si avvia un thread che, ad ogni secondo, invia una richiesta di acquisizione dati relativa alla frequenza cardiaca. Quando la simulazione viene interrotta, si interrompe l'esecuzione del thread e si comunica al server il termine della sessione real time. Inoltre, si effettuano le medesime operazioni effettuate dal controller relativo alla simulazione settimanale, ma con una differenza per quanto riguarda l'elaborazione del feedback; il quale viene elaborato sulla sessione corrente e non sull'arco di una settimana.
Alla ricezione della frequenza cardiaca, vengono svolte le istruzioni presenti di seguito.

```
1  EHealthDataListener listener=new EHealthDataListener() {
2    @Override
3    public void update(BluetoothEHealthData.Type type, IData value, String
         name, String address) {
4      if(simulation.isRunning()) {
5        synchronized (lock) {
6          bpmSum += ((HeartRate) value).getValue();
7          count++;
8        }
9
10       view.heartrate(((HeartRate) value).getValue());
11     }
12   }
13
14   @Override
15   public void update(BluetoothEHealthData.Type type, List<IData> values,
         String name, String address) {}
```

```
16
17    @Override
18    public void dataNotAvailable(BluetoothEHealthData.Type type, String name,
          String address) {}
19
20    @Override
21    public void connectionClosed(String name, String address) {}
22  };
```

Ogni qualvolta che si riceve un valore relativo alla frequenza cardiaca, si somma ai valori precedenti e si incrementa il contatore che tiene conto delle volte in cui è stata ricevuta una misurazione.
Alla richiesta di dati da parte del modello, si procede come illustrato di seguito.

```
1  simulation.addObserverRequestData(new ObserverRequestData() {
2      @Override
3      public JSONObject requestData() {
4        JSONObject json=new JSONObject();
5        try{
6          double average;
7          synchronized (lock){
8            if(count!=0) {
9              average = bpmSum / count * 1.0;
10           }else{
11             average=0;
12           }
13           bpmSum=0;
14           count=0;
15         }
16         json.put("heartrate",average);
17
18         synchronized (lockMeal){
19           mealRealtime=new JSONArray();
20           if(meals.size()>0){
21             ArrayList<FoodInfo> foods=new ArrayList<>();
22             for(Meal meal:meals){
23               foods.addAll(meal.getFoods());
24               for(FoodInfo tmp : foods){
25                 mealRealtime.put(tmp.toJSON());
26                 totalFood++;
27                 totalGlycemicLoad=tmp.getGlycemicLoad();
28               }
29             }
30             json.put("meal",foods);
31
32             meals.clear();
33           }
34         }
35       }catch(JSONException e){
36         e.printStackTrace();
37       }
```

```
38      return json;
39    }
40  });
```

Ogni qualvolta che viene richiesto un dato (1 volta al minuto), si calcola la media della frequenza cardiaca, la quale viene passata al modello. Inoltre, al momento della richiesta di acquisizione dati da parte del modello, si controlla se sono stati effettuati dei pasti nel minuto precedente. Come per la simulazione settimanale, per ogni pasto si tiene traccia del carico glicemico totale e del numero di alimenti ingeriti.
Di seguito è presente il listato con le operazioni svolte ogni qualvolta che il modello fornisce dei dati.

```
1  simulation.addObserverData(new ObserverDataSimulation() {
2      @Override
3      public void update(final JSONObject data) {
4        try {
5          time++;
6          final long step = data.getLong("step");
7          final double glycemia=data.getDouble("glycemia");
8          final double insulin=data.getDouble("insulin");
9          final double glucagon=data.getDouble("glucagon");
10         final double calories=data.getDouble("calories");
11         double boluses=0;
12
13         for(GlycemiaMealInfo tmp:glycemiaMeal){
14           if(simulation.getSteps()>=tmp.getStep()&simulation.getSteps()<=
                 (tmp.getStep()+120)){
15             if(tmp.getGlycemiaMax().second<data.getDouble("glycemia")){
16               tmp.setGlycemiaMax(new
                     android.support.v4.util.Pair<>(simulation.getSteps(),
                     glycemia));
17               tmp.setGlycemiaPostPrandial(new
                     android.support.v4.util.Pair<>(simulation.getSteps(),
                     glycemia));
18             }
19           }
20
21           if(simulation.getSteps()>=(tmp.getStep()+120)&&
                 imulation.getSteps()<=(tmp.getStep()+240)){
22             if(tmp.getGlycemiaPostPrandial().second>
                   data.getDouble("glycemia")){
23               tmp.setGlycemiaPostPrandial(new
                     android.support.v4.util.Pair<>(simulation.getSteps(),
                     glycemia));
24             }
25           }
26         }
27
28         if(data.has("boluses")){
```

```
29          glycemiaMeal.add(new GlycemiaMealInfo((int)step,new
                android.support.v4.util.Pair<>(step,glycemia),new
                android.support.v4.util.Pair<>(step,glycemia),new
                android.support.v4.util.Pair<>(step,glycemia)));
30          boluses=data.getDouble("boluses");
31        }
32
33        totalCalories+=data.getDouble("calories");
34
35        SimpleDateFormat format=new SimpleDateFormat("YYYY-MM-dd HH:mm:ss");
36        JSONObject params=new JSONObject();
37        params.put("username",username);
38        params.put("id_session",username+"_"+date);
39        params.put("date",format.format(Calendar.getInstance().getTime()));
40        params.put("glycemia",glycemia);
41        params.put("glucagon",glucagon);
42        params.put("insulin",insulin);
43        params.put("liver_glycogen",data.getDouble("liver_glycogen"));
44        params.put("muscle_glycogen",data.getDouble("muscle_glycogen"));
45        params.put("calories",calories);
46        params.put("meal",mealRealtime);
47
48        HttpRequest.getInstance(context).sendJSONRequest(Request.Method.POST,
              ipaddress+"/send-data", params, new
              Response.Listener<JSONObject>() {
49          @Override
50          public void onResponse(JSONObject response) {
51          }
52        }, new Response.ErrorListener() {
53          @Override
54          public void onErrorResponse(VolleyError error) {
55            System.out.println(error.getMessage());
56            error.printStackTrace();
57          }
58        });
59
60        view.calories(calories);
61        view.boluses(boluses);
62        view.newData(glycemia,glucagon,insulin,(int)step);
63      }catch(JSONException e){
64        e.printStackTrace();
65      }
66    }
67  });
68  %
```

Alla ricezione dei dati forniti dal modello, quest'ultimi vengono estratti e utilizzati. Se sono presenti delle calorie, il paziente ha effettuato un pasto e, pertanto, viene mostrato a quest'ultimo la quantità di boli di insulina da assumere.
Inoltre, come visibile dal listato, ogni qualvolta che si ricevono dei dati, que-

st'ultimi vengono inviati al server e mostrati all'assistito attraverso il livello di presentazione.
Come per il caso della simulazione, ai fini dell'elaborazione da parte del sistema di regole, si effettuano tutte le operazioni necessarie per rilevare lo stato della glicemia nel corso di un pasto (glicemia pre/post prandiale e picco massimo).

Capitolo 7

Sviluppi futuri

Sebbene il sistema realizzato contenga tutte le principali funzionalità richieste, vi sono ampi margini di miglioramento. All'interno di questo capitolo sono presenti gli aspetti su cui è possibile migliorare il sistema.

7.1 Accuratezza e validazione del modello

Per rendere possibile l'utilizzo del sistema realizzato, è necessario garantire un adeguato livello di accuratezza del modello adoperato per la simulazione del metabolismo del glucosio.
Nel corso dell'attività è stato effettuato un confronto con un simulatore approvato dall'FDA (Food and Drugs Administration). È emersa una notevole discrepanza tra i dati del modello validato e quello utilizzato per la realizzazione del sistema.
Affinché il sistema realizzato possa essere utilizzato, si evince la necessità di incrementare l'accuratezza del modello attualmente in uso.

7.2 Gamification

Il self-Management dipende molto dalla volontà della persona interessata e dall'educazione che esso riceve su tale argomento. Per incentivare il paziente a utilizzare l'applicazione, può risultare utile introdurre gli aspetti tipici della gamification. Con tale termine si intende l'utilizzo di tecniche di game design

ed elementi tipici dei giochi, in contesti esterni a quello ludico. L'adozione di queste tecniche è finalizzata ad aumentare il coinvolgimento dell'utente in attività che potrebbero risultare noiose.

7.3 Educazione self-Management

Come illustrato nel paragrafo precedente e nei capitoli introduttivi di questa trattazione, una componente fondamentale per l'empowerment del paziente è l'educazione che esso riceve. A tal fine, per incrementare le conoscenze dell'assistito, all'interno dell'applicazione si potrebbero inserire dei video esemplificativi su come effettuare una misurazione della pressione arteriosa e della glicemia. Inoltre, si potrebbe aggiungere del materiale informativo con lo scopo di illustrare i rischi del diabete e i sintomi da tenere maggiormente sotto controllo.

7.4 Miglioramento sessione real time

Allo stato attuale, la sessione real time non fa altro che simulare, minuto per minuto, il possibile valore glicemico del paziente. Al fine di migliorare tale funzionalità, si potrebbe inserire una componente predittiva, la quale simulerà una finestra temporale che verrà costantemente corretta con il proseguire della sessione. In questo modo, qualora la glicemia dell'assistito dovesse avvicinarsi alla soglia dell'ipoglicemia o dell'iperglicemia, si verrebbe a creare un sistema predittivo in grado di notificare sia il medico che il paziente.

7.5 Elaborazione del feedback

Allo stato attuale, per l'elaborazione del feedback, il sistema considera le preferenze inserite manualmente dal paziente ogni qualvolta che effettua una sessione o una simulazione.
Data la sempre maggiore potenza computazionale dei device mobili e la realizzazione di piattaforme hardware che forniscono supporto a soluzioni che adottano l'intelligenza artificiale, è possibile migliorare questa funzionalità grazie all'adozione delle reti neurali.
Si potrebbero impiegare delle reti neurali in grado di analizzare, nel tempo, i

pasti effettuati e le attività fisiche svolte dall'assistito nel corso delle diverse sessioni real time; al fine di inferire le preferenze senza il bisogno che l'assistito le inserisca ad ogni simulazione o sessione.
Naturalmente, parte dell'accuratezza della soluzione dipenderà dall'utilizzo da parte dell'assistito dell'applicazione. Maggiore sarà l'uso della funzionlità di simulazione, maggiore sarà la precisione del sistema nel dedurre le preferenze dell'assistito.

7.6 Comunicazione con un professionista sanitario

Al momento il sistema consente la comunicazione da parte del medico curante con l'assistito attraverso l'invio di notifiche push. Al fine di migliorare l'attuale metodo di comunicazione, si potrebbe realizzare un canale di comunicazione bidirezionale tra i due attori coinvolti; offrendo la possibilità anche al paziente di comunicare con il professionista sanitario.

7.7 Sicurezza

Sebbene non sia stato affrontato all'interno di questa tesi, uno degli aspetti su cui è necessario lavorare, qualora si volesse rendere disponibile l'applicazione, è la sicurezza. In tale ambito la sicurezza riveste un ruolo ancora più importante; data la sensibilità dei dati utilizzati e trasferiti. Bisogna garantire la sicurezza dei dati sia in fase di trasferimento, che di memorizzazione.

Conclusioni

Il lavoro svolto ha permesso di approfondire le tematiche del mobile Health, del self-Management e dell'Internet of Things. Questi aspetti rappresentano la base su cui è stato realizzato il sistema illustrato all'interno di questra trattazione.
La volontà del sistema realizzato, non è quello di fornire al paziente un semplice diario delle misurazioni, ma vuole fornire gli strumenti necessari per permettere di effettuare delle simulazioni per un breve periodo di tempo e notare i possibili cambiamenti al variare degli accorgimenti adottati sul regime alimentare e sull'attività fisica svolta.
Al fine di consentire all'assistito di effettuare delle scelte sullo stile di vita da adottare, in modo corretto, minimizzando la possibilità di commettere degli errori dovuti alle scarse conoscenze mediche, è stato realizzato un meccanismo attraverso il quale il medico definisce delle regole. Quest'ultime si occupano di analizzare i dati prodotti dalla simulazione e/o dalla sessione real time e forniscono un feedback al paziente; sotto forma di possibili regimi alimentari e attività fisiche da seguire. Per la realizzazione di questa porzione del sistema, l'utilizzo di un linguaggio che adotta la logica del primo ordine come Prolog è risultato fondamentale; in quanto consente di definire la struttura logica del problema ed evita allo sviluppatore di soffermarsi sui dettagli implementativi per la realizzazione della soluzione.
Nella realizzazione del sistema, sebbene quest'ultimo non sia pronto per essere reso disponibile al pubblico, data la mancanza di aspetti relativi alla sicurezza e alla validazione del modello che effettua la simulazione, i risultati ottenuti si possono considerare soddisfacenti. Trattandosi di un sistema che può permettere alla struttura sanitaria di monitorare più pazienti, vi è la necessita che il sistema sia scalabile. Grazie alla possibilità di effettuare la simulazione e l'analisi dei dati sul device dell'assistito, questo requisito è ampiamente soddisfatto; in quanto l'unica complessità presente lato server è la generazione

del sistema di regole sotto forma di file Prolog. Inoltre, grazie all'adozione di NodeJs (con l'ausilio del web framework Express) e RethinkDB, il server risulta essere performante e in grado di gestire molteplici richieste in modo concorrente.
Nella realizzazione del sistema sono state create delle librerie che possono essere utilizzate per realizzare sistemi per la gestione di malattie croniche. Le librerie realizzate consentono di gestire la trasmissione dei dati tra due device via Bluetooth e, inoltre, forniscono gli strumenti necessari per consentire lo scambio dei dati, nel corso di una simulazione, con qualsiasi modello ad agenti. Lato server non è stato realizzato nessun modulo per NodeJs per lo sviluppo di questo tipo di sistemi; in quanto, grazie alla tecnologia utilizzata, in grado di fornire un elevato livello di astrazione, la complessità è risultata essere pressoché nulla nella.
Si tiene a segnalare la capacità, grazie a un livello di maturazione del linguaggio Java utilizzato da Android, di realizzare un'applicazione adottando diversi paradigmi di programmazione. Nell'applicazione realizzata, infatti, oltre al paradigma ad oggetti, è stato possibile utilizzare, attraverso il framework MASON, il paradigma ad agenti. Inoltre, è stato possibile utilizzare un interprete Prolog per la porzione del sistema relativa alla generazione del feedback.
Sulla base dei risultati ottenuti, è possibile affermare che il prodotto realizzato sia utile per lo sviluppo futuro di questo genere di sistemi.

Riferimenti

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, pp. 2347–2376, Fourthquarter 2015.

[2] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: The internet of things architecture, possible applications and key challenges," in *2012 10th International Conference on Frontiers of Information Technology*, pp. 257–260, Dec 2012.

[3] Microsoft, "Weka costruisce un frigorifero intelligente per aiutare a salvare vite." `https://www.microsoft.com/it-it/internet-of-things/customer-stories#healthcare&wekasolutions`.

[4] S. Latif, R. Rana, J. Qadir, A. Ali, M. A. Imran, and M. S. Younis, "Mobile health in the developing world: Review of literature and lessons from a case study," *IEEE Access*, vol. 5, pp. 11540–11556, 2017.

[5] M. Hardinge, H. Rutter, C. Velardo, S. A. Shah, V. Williams, L. Tarassenko, and A. Farmer, "Using a mobile health application to support self-management in chronic obstructive pulmonary disease: a six-month cohort study," in *BMC Med. Inf. Decision Making*, 2015.

[6] "Mobile-enabled rapid cardiovascular screening improves health care for rural patients in china." `https://www.qualcomm.com/media/documents/files/china-heart-health.pdf`.

[7] A. Bourouis, A. Zerdazi, M. Feham, and A. Bouchachia, "M-health: Skin disease analysis system using smartphone's camera," *Procedia Computer Science*, vol. 19, pp. 1116 – 1120, 2013. The 4th International Conference

on Ambient Systems, Networks and Technologies (ANT 2013), the 3rd International Conference on Sustainable Energy Information Technology (SEIT-2013).

[8] A. C. Dammert, J. C. Galdo, and V. Galdo, "Preventing dengue through mobile phones: Evidence from a field experiment in peru," *Journal of Health Economics*, vol. 35, pp. 147 – 161, 2014.

[9] J. D. Piette, H. Datwani, S. Gaudioso, S. M. Foster, J. Westphal, W. Perry, J. Rodríguez-Saldaña, M. O. Mendoza-Avelares, and N. Marinec, "Hypertension management using mobile technology and home blood pressure monitoring: Results of a randomized trial in two low/middle-income countries," *Telemedicine and e-Health*, vol. 18, no. 8, pp. 613–620, 2012. PMID: 23061642.

[10] A. Bakshi, P. Narasimhan, J. Li, N. Chernih, P. K. Ray, and R. MacIntyre, "mhealth for the control of tb/hiv in developing countries," in *2011 IEEE 13th International Conference on e-Health Networking, Applications and Services*, pp. 9–14, June 2011.

[11] U. Strandbygaard, S. F. Thomsen, and V. Backer, "A daily sms reminder increases adherence to asthma treatment: A three-month follow-up study," *Respiratory Medicine*, vol. 104, no. 2, pp. 166 – 171, 2010.

[12] T. M. da Costa, P. L. Salomão, A. S. Martha, I. T. Pisa, and D. Sigulem, "The impact of short message service text messages sent as appointment reminders to patients' cell phones at outpatient clinics in são paulo, brazil," *International Journal of Medical Informatics*, vol. 79, no. 1, pp. 65 – 70, 2010.

[13] J. E. Aikens, K. Zivin, R. Trivedi, and J. D. Piette, "Diabetes self-management support using mhealth and enhanced informal caregiving," *Journal of Diabetes and its Complications*, vol. 28, no. 2, pp. 171 – 176, 2014.

[14] C. Hacks, "Piattaform e-health v 2.0." `https://www.cooking-hacks.com/documentation/tutorials/ehealth-biometric-sensor-platform-arduino-raspberry-pi-medical`.

[15] F. Vasini, "Healthpi library process report." `http://apice.unibo.it/xwiki/bin/view/Tirocini/ehealth14vasini`.

[16] Google, "Protocol buffers." `https://developers.google.com/protocol-buffers/`.

[17] S. Luke, "Multiagent simulation and the mason library." `https://cs.gmu.edu/~eclab/projects/mason/manual.pdf`.

[18] F. D. Angeli, "Self-management di malattie croniche in sistemi di mobile health: Sviluppo di un modello agent-based per casi di diabete," Master's thesis, Università di Bologna, 3 2015.

[19] I. Khajenasiri, A. Estebsari, M. Verhelst, and G. Gielen, "A review on internet of things solutions for intelligent energy control in buildings for smart city applications," *Energy Procedia*, vol. 111, pp. 770 – 779, 2017. 8th International Conference on Sustainability in Energy and Buildings, SEB-16, 11-13 September 2016, Turin, Italy.

[20] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[21] Z. Yang, Y. Yue, Y. Yang, Y. Peng, X. Wang, and W. Liu, "Study and application on the architecture and key technologies for iot," in *2011 International Conference on Multimedia Technology*, pp. 747–751, July 2011.

[22] S. L. Aronoff, K. Berkowitz, B. Shreiner, and L. Want, "Glucose metabolism and regulation: Beyond insulin and glucagon," *Diabetes Spectrum*, vol. 17, no. 3, pp. 183–190, 2004.

[23] N. Cho, J. Shaw, S. Karuranga, Y. Huang, J. da Rocha Fernandes, A. Ohlrogge, and B. Malanda, "Idf diabetes atlas: Global estimates of diabetes prevalence for 2017 and projections for 2045," *Diabetes Research and Clinical Practice*, vol. 138, pp. 271 – 281, 2018.

[24] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, pp. 80–83, Nov 2010.

[25] U. Varshney, "Mobile health: Four emerging themes of research," *Decision Support Systems*, vol. 66, pp. 20 – 35, 2014.

[26] Angular, "Angular documentation." https://angular.io/docs.

[27] A. Gerber, "Connecting all the things in the internet of things." https://www.ibm.com/developerworks/library/iot-lp101-connectivity-network-protocols/index.html, 5 2017.

[28] Microsoft, "Promozione di prassi ospedaliere avanzate per l'igiene delle mani con soluzioni iot." https://www.microsoft.com/it-it/internet-of-things/customer-stories#healthcare&gojoindustries.

[29] Android, "Documentazione bluetooth android." https://developer.android.com/guide/topics/connectivity/bluetooth.

[30] Android, "Documentazione volley android." https://developer.android.com/training/volley/.

[31] NodeJs, "Documentazione nodejs." https://nodejs.org/en/docs/.

[32] Arduino, "Arduino." https://www.arduino.cc/.

[33] RaspberryPi, "Raspberrypi." https://www.raspberrypi.org/.

[34] Bluecove, "Bluecove." http://www.bluecove.org/.

[35] BlueZ, "Bluez." http://www.bluez.org/.

[36] Nokia, "Nokia blood pressure monitor." https://health.nokia.com/mx/en/blood-pressure-monitor.

[37] iHealth, "ihealth." https://ihealthlabs.eu/it/.

[38] Beurer, "Beurer." https://www.beurer.com/web/it/index.php.

[39] Qardio, "Qardio." https://www.getqardio.com/it.

[40] Koogeek, "Koogeek." https://www.koogeek.com/.

[41] Omron, "Omron." https://omron.it/it/home.

[42] OneTouch, "Onetouch." https://www.onetouch.it/.

[43] Contour, "Contour." https://www.diabetes.ascensia.it/.

[44] Diabnext, "Gluconext." https://diabnext.com/gluconext/.

[45] Smashicons, "Icon made by Smashicons from Flaticon." https://www.flaticon.com/.

[46] Freepik, "Icon made by Freepik from Flaticon." https://www.flaticon.com/.

[47] monkik, "Icon made by monkik from Flaticon." https://www.flaticon.com/.