Alma Mater Studiorum · Università di
Bologna Campus di Cesena

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Triennale in Ingegneria Elettronica, Informatica e Telecomunicazioni

La programmazione logica per l'Internet of Things: il caso di studio del frigorifero intelligente

Tesi in Sistemi Distribuiti

Relatore:
Andrea Omicini

Presentata da:
Nicola Atti

Correlatore:
Giovanni Ciatto

Sessione II
Anno Accademico 2016/2017

Introduzione

Ultimamente le applicazioni relative al paradigma dell'Internet of Things richiedono sempre di più una forma di conoscenza e comprensione dell'ambiente in cui sono immerse, in poche parole è necessario che gli oggetti dell'IoT diventino intelligenti.
L'oggetto di questa tesi è mostrare come il paradigma della programmazione logica si possa coniugare con quello dell'Internet of Things tramite un caso di studio per un frigorifero intelligente, in grado di conoscere i suoi contenuti e di ragionare su di essi, sottolineando i vantaggi dell'approccio logico e le eventuali modifiche agli standard dell'IoT visti fino ad ora.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

L'Internet of Things

1.1 Cos'è l'Internet of Things?

L'Internet of Things, o abbreviato IoT, è un paradigma che ha rapidamente guadagnato terreno nello scenario delle moderne telecomunicazioni wireless.
In questo paradigma, gli oggetti della vita di tutti i giorni o dell'ambiente che ci circonda, chiamati anche "things" o "objects", sono dotati di capacità computazionali e comunicative, e sono collegati tramite un framework di comunicazione [1].
Dalla data della sua nascita, fino ai giorni nostri, questo paradigma ha avuto un crescente impatto sulla vita di tutti i giorni, sia dal punto di vista dell'utente privato, sia da un punto di vista strettamente legato al mondo del business.
L'Internet of Things nasce ufficialmente nel 2005, quando l'International Telecomunication Unit (ITU) mostra il suo interesse per il lavoro svolto da EPCglobal ed EAN UCC(European Article Numbering Uniform Code Council) per portare il concetto di Electronic Product Code (EPC) all interno delle realtà industriali.
L'obiettivo di EPC era quello di referenziare a livello di software ogni prodotto elettronico presente nella vita reale, e di inserirlo in una catena di

comunicazione, creando cosi una connessione tra il mondo reale e quello digitale. L'unione di questi due mondi ha permesso l'aggiunta di una nuova caratteristica della connettività, alle già esistenti "anywhere" e "anytime", cioè quella dell "anything" che introduce le interazioni del tipo thing-to-thing o machine-to-machine, oltre a quelle già esistenti person-to-person e person-to-machine [4].

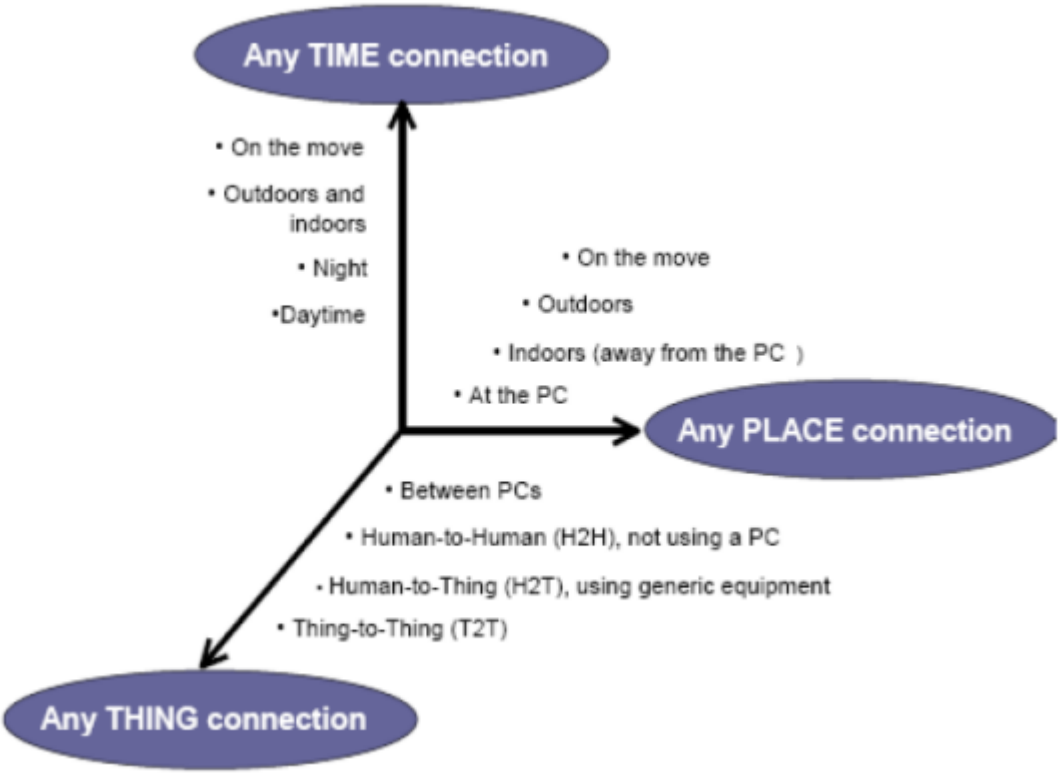

Figura 1.1: Principi dell'internet of Things.

Il concetto chiave dietro l'IoT è "la presenza pervasiva intorno a noi di una varietà di cose o oggetti, come tag RFID (Radio Frequency Identification), sensori, attuatori, dispositivi mobile, etc. - i quali attraverso unici schemi di indirizzamento sono in grado di interagire gli uni con gli altri e cooperare

con i loro vicini" [3].
Il paradigma dell'Internet of Things è in costante evoluzione, grazie al continuo miglioramento delle tecnologie a cui si appoggia, fino ad arrivare ai giorni nostri, in cui stiamo assistendo alla nascita di una nuova forma.

1.2 Verso l'Internet of Intelligent Things

Le potenzialità offerte dall'Internet of Things permettono di sviluppare un enorme quantità di applicazioni, di cui però solo una piccola parte è disponibile all'interno della nostra società odierna, come ad esempio: applicazioni wearable per il monitoraggio della salute o per la gestione delle attività di fitness, applicazioni per rintracciare oggetti o per gestire i consumi di energia all'interno della propria abitazione.

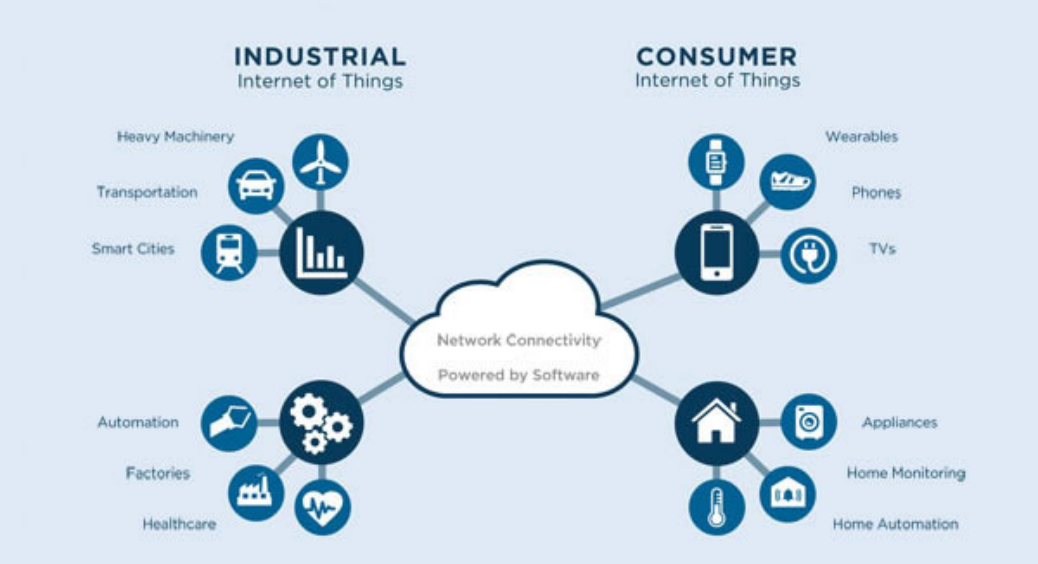

Figura 1.2: Domini applicativi dell'Internet of Things.

Le applicazioni sviluppate fino ad ora spesso si limitano a monitorare e a raccogliere dati sull'ambiente in cui sono immerse e lasciano all'utente

il compito di utilizzare i dati raccolti, ma ultimamente ci si sta muovendo sempre più verso l'idea di avere oggetti intelligenti, in grado di comprendere l'ambiente che li circonda e prendere decisioni autonomamente in base ai dati percepiti, questa visione ci porta verso l'Internet of Intelligent Things [3].

Fintanto che continuerà la rapida espansione dei dispositivi e dei sensori legati all'Internet of Things, il volume dei dati da essi prodotto aumenterà esponenzialmente e l'intelligenza artificiale (AI), che ha ormai raggiunto un punto in cui, accoppiata al paradigma dell'Internet of Things, può fornire un prezioso aiuto nel realizzare compiti che sono ancora svolti dagli esseri umani.
Uno degli strumenti più usati per la programmazione di intelligenza artificiale è la programmazione logica, che consiste nell'uso di dichiarazioni logiche per descrivere che cos'è vero all'interno di un dato dominio, in questo modo è possibile fornire conoscenza alle applicazioni.
Nella programmazione logica la computazione è data dalla deduzione, le risposte alle domande sono ottenute tramite la deduzione delle conseguenze logiche definite dal programma e dalle possibili regole in esso contenute, che servono a guidare l'interprete logico per raggiungere le giuste conclusioni.

In questo capitolo sono stati descritti brevemente alcuni concetti chiave alla base dell'Internet of Things, e di come questa visione iniziale si sta evolvendo sempre più verso una forma di intelligenza, grazie al suo inevitabile collegamento con l'intelligenza artificiale.

Nel prossimo capitolo si entrerà più nel dettaglio riguardo alla programmazione logica, soffermandosi soprattutto sul linguaggio proposto per questo lavoro: tuProlog.

Capitolo 2

La programmazione logica

2.1 Origini della programmazione logica

Largamente basato sulla logica formale, ogni programma scritto in un linguaggio di programmazione logico è dato da un insieme di frasi in forma logica, che esprimono fatti e regole riguardo ad un dominio specifico.

Il paradigma di programmazione logico ha la sue radici nella dimostrazione automatica dei teoremi, dal quale ha ereditato la nozione di deduzione.
La nascita di questo paradigma è il risultato di una lunga storia che per la maggior parte del suo corso si è basata sulla logica, e solo più tardi nell'ambito della computer science.
La programmazione logica si basa sulla sintassi della logica del primo ordine, originariamente proposta nella seconda metà del diciannovesimo secolo da Gottlob Frege, e in seguito modificata nella forma usata attualmente da Giuseppe Peano e Bertrand Russel.
Usando il metodo di risoluzione (introdotto nel 1965 da Aland Robinson) si possono provare teoremi della logica del primo ordine, ma era necessario un altro passo prima di poter osservare come si potesse computare all'interno di questo framework.
Questo fu infine ottenuto nel 1974 da Robert Kowalski nel suo saggio [5] nel

quale introdusse programmi logici con una forma di risoluzione ristretta. La differenza tra questa nuova forma di risoluzione e quella proposta da Robinson era che la sintassi era più ristretta, ma dimostrare aveva un effetto collaterale nella forma di una sostituzione soddisfacente, questa sostituzione può essere vista come il risultato di una computazione e conseguentemente alcune formule logiche possono essere viste come programmi.

In contemporanea a questo lavoro, Alain Colmerauer assieme ai suoi colleghi lavorava ad un linguaggio di programmazione per l'elaborazione del linguaggio naturale, basato sulla dimostrazione automatica dei teoremi.
Questo lavoro portò in definitiva alla creazione di Prolog nel 1973.
Prolog nacque dunque come linguaggio di programmazione per l'elaborazione del linguaggio naturale, ma subito dopo si scoprì che poteva essere usato anche come linguaggio general purpose.

Il paradigma di programmazione logico ha influenzato un gran numero di sviluppi in informatica.
Già negli anni settanta portò alla creazione di database deduttivi, che estendono i database relazionali dotandoli di capacità deduttive e più recentemente condusse alla programmazione logica a vincoli, che realizza un approccio generale alla computazione in cui il processo di programmazione è limitato a una generazione di vincoli (requisiti) e ad una loro soluzione, e alla programmazione logica induttiva, un approccio logico al machine learning.

Questa descrizione della storia della programmazione logica e in particolare di Prolog mostra chiaramente come le sue radici siano ben salde nel campo della logica e della deduzione automatica.
Infatti, Colmerauer e Roussel scrissero in [6] "Non vi è alcun dubbio che Prolog sia essenzialmente un deduttore di teoremi 'a la Robinson'. Il nostro contributo è stato quello di trasformare tale deduttore di teoremi in un linguaggio di programmazione."

L'origine di questo paradigma ne ha probabilmente ostacolato la sua accettazione all'interno della computer science negli anni in cui la programmazione imperativa otteneva slancio dalla creazione di linguaggi come Pascal e C e in cui la comunità dell'intelligenza artificiale aveva già adottato Lisp come linguaggio predefinito.

2.2 Elementi chiave della programmazione logica, il caso di tuProlog

2.2.1 Le basi di Prolog

Il paradigma di programmazione logico si differenzia sostanzialmente dagli altri paradigmi di programmazione.
Se ridotto ai suoi temi essenziali, può essere sintetizzato nelle seguenti tre caratteristiche:

- La computazione ha luogo sul dominio di tutti i termini definiti su un alfabeto "universale".
- Dei valori sono assegnati alle variabili tramite sostituzione, chiamate "most general unifiers". Questi valori potranno contenere variabili, dette variabili logiche.
- Il controllo è fornito da un singolo meccanismo, il backtracking automatico.

Inoltre la programmazione logica supporta la programmazione dichiarativa.
Un programma dichiarativo ammette due tipi di interpretazione, la prima, detta interpretazione procedurale, esprime dove ha luogo la computazione, mentre la seconda, chiamata interpretazione dichiarativa, si concentra sul cosa viene computato.
In poche parole l'approccio procedurale si interessa del metodo. Un programma è visto come la descrizione di un algoritmo da eseguire.

L'approccio dichiarativo invece si interessa del significato, in questo caso un programma è visto come una formula, sulla quale ragionare sulla sua correttezza senza dover conoscere alcun meccanismo di computazione sottostante. Questa doppia interpretazione è responsabile del duplice uso della programmazione logica, come formalismo per la programmazione e per rappresentare la conoscenza.

Un'altra caratteristica importante della programmazione logica è il suo supporto della programmazione interattiva.
Un utente potrà scrivere un singolo programma ed interagirci attraverso query di interesse, dalle quali vengono prodotte delle risposte. I sistemi Prolog supportano questo tipo di interazione e forniscono semplici metodi per calcolare le multiple soluzioni alla query sottoposta, per sottomettere un'altra query, e per tenere traccia dell'esecuzione tramite la creazione di vari punti di controllo (check point), sempre all'interno delel stesso "loop di interazione".

I termini base della programmazione logica

Nella programmazione logica le azioni atomiche sono equazioni fra i termini (espressioni arbitrarie), che vengono eseguite tramite il processo di unificazione che cerca di risolverle.
Durante tale processo, alle variabili sono associati dei valori che possono essere termini arbitrari, infatti tutte le variabili sono dello stesso tipo, che consiste nell'insieme di tutti i termini.
Per entrare più nello specifico, si possono definire i seguenti simboli:

- variabili x,y,z,...
- simboli delle funzioni
- parentesi, "(" e ")"
- virgole

In particolare, ogni simbolo delle funzioni ha una sua arietà, che definisce il numero di argomenti ad essa associati. Una funzione con arietà zero è detta costante.
I termini sono definiti come segue:

- una variabile è un termine
- se f è il simbolo di una funzione n-aria e t1,...,tn sono termini, allora f(t1,...,tn) è un termine.

In particolare ogni costante è un termine, e i termini senza variabili sono comunemente chiamati "ground terms".

Nella programmazione logica non si assume un alfabeto specifico. Un semplice esempio di questa affermazione è dato dai simboli che normalmente usiamo per le operazioni matematiche (quali il +,-,*,.. etc) i quali, nella programmazione logica non assumono alcun significato a priori. In questo modo nessun tipo viene assegnato ai termini, quindi non vi è alcuna distinzione tra i vari simboli utilizzati.

Il meccanismo di sostituzione e il Most General Unifier

A differenza di altri linguaggi, nella programmazione logica le variabili non sono inizializzate.
Ad un dato momento nell'esecuzione di un programma esiste un numero finito di variabili che possono essere inizializzate, queste sono variabili a cui, nella computazione considerata, era già stato assegnato un valore.
Siccome questi valori sono termini, si può parlare di sostituzione, cioè un numero finito di mappature da variabili a termini tali che nessuna variabile sia mappata a se stessa. Quindi la sostituzione fornisce informazioni su quali variabili sono inizializzate.
Una sostituzione è denotata dalla forma (x1/t1, . . . , xn/tn).
Questa notazione implica che x1, . . . , xn sono variabili distinte e t1, . . . , tn sono termini e che nessun termine tk eguaglia la variabile xk.

Si dirà che la sostituzione (x1/t1, . . . , xn/tn) lega la variabile xk con il termine tk. Usando una sostituzione si può valutare un termine, questo processo di valutazione è definito come l'applicazione di una sostituzione ad un termine ed è il risultato di rimpiazzi simultanei di ogni variabile presente nel dominio della sostituzione con il termine corrispondente.
Ad esempio la sostituzione (x/f(z), y/g(z)) al termine h(x, y), fornisce il termine h(f(z), g(z)).
Qui la variabile x è stata rimpiazzata dal termine f(z) e la variabile y dal termine g(z).
Allo stesso modo si può definire la sostituzione ad un atomo, una query o una clausola.

Scrivendo "x = a" si assegna la costante 'a' alla variabile x. Siccome nella programmazione logica l'uguaglianza "=" è simmetrica, si ottiene lo stesso risultato ponendo a = x.
Scrivendo x = f(y) si assegna il termine f(y) alla variabile x, ma visto che f(y) è un termine con una variabile, si sta assegnando alla variabile x un'espressione con una variabile al suo interno.
Una variabile che compare in un valore assegnato ad una variabile si chiama variabile logica, come la y dell'esempio precedente.
Infine scrivendo f(y) = f(g(a)) si assegna il termine g(a) alla variabile y.

L'unificazione è il processo di risoluzione di un'equazione fra termini nel modo meno vincolante, e la sostituzione risultante, se essa esiste, è chiamata "Most General Unifier" (MGU).

La composizione di un programma logico e il suo meccanismo di esecuzione

Se p è una relazione n-aria e t1, . . . , tn sono termini, allora chiameremo p(t1, . . . , tn) un atomo. Quando n=0, p viene chiamato simbolo proposizionale.

Dato un simbolo relazionale n-ario p e degli atomi A1, . . . , Ak la forma p(x1, . . ., xn) : - A1, . . ., Ak. viene definita come una clausola semplice.
La parte p(x1, . . .,xn) viene chiamata testa della clausola, mentre la parte A1, . . ., Ak viene chiamata corpo. Il punto alla fine della definizione della clausola serve per segnalare al compilatore che ci si trova alla fine della dichiarazione di tale clausola.
Le clausole si dividono in multiple categorie a seconda dell'arietà della testa e del corpo.

- Nel caso di n=1 si parla di clausola definita
- Nel caso di n=1 e m=0 si parla di clausola unitaria
- Nel caso di n=0 si parla di obbiettivo definito

Una "Horn clause" può essere sia una clausola definita, sia un obbiettivo definito.
All'interno di un programma logico una clausola definita viene chiamata "rule", una clausola unitaria "fact" mentre un obbiettivo definito viene semplicemente chiamato "goal".
Un programma logico è sostanzialmente composto da clausole Horn, quindi da facts e rules.

Come accennato in precedenza, le clausole possono essere interpretate sia in modo procedurale (per dimostrare la testa bisogna dimostrare il corpo), sia in modo dichiarativo (l'affermazione nella testa è vera se è vera l'affermazione nel corpo).
Permettendo a multiple clausole di avere lo stesso simbolo relazionale nella testa si introduce una forma di non-determinismo, che in programmazione logica è chiamato "don't know non-determininsm".
Questa forma di non-determinismo è mantenuta in programmazione logica considerando tutte le computazioni che possono essere generate dalle multiple clausole, mantenendo quelle che portano ad un successo. Viene definito "don't know" poichè non si sa a priori quali clausole porteranno al successo.

In Prolog questo processo viene reso deterministico ordinando le clausole nell'ordine in cui sono scritte e utilizzando il backtracking automatico per riprendersi da eventuali fallimenti.
Se ad un determinato passaggio si incappa in un fallimento, la computazione esegue backtracking fino all'ultimo punto di scelta (cioè un punto nella computazione in cui sono state selezionate una o più clausole applicabili) e si seleziona la clausola successiva.
Se la clausola selezionata è l'ultima, la computazione fa backtracking all'ultimo punto di scelta, se non vi sono altri punti a cui ritornare, si incorre in un fallimento.
Il backtracking comporta il ripristino dello stato precedente, quindi tutti i cambiamenti effettuati dalla creazione del punto di scelta verso cui si sta ritornando sono cancellati.

L'uso della negazione

La programmazione logica fornisce anche operatori in grado di utilizzare una situazione di fallimento di una query e trasformarla in un successo: l'operatore not.
In questo caso una query scritta come "not Q." ha successo se "Q." fallisce.

2.2.2 tuProlog

tuProlog è un sistema Prolog leggero per applicazioni e infrastrutture distribuite, progettato intenzionalmente attorno ad un core minimale, per essere configurato sia staticamente che dinamicamente caricando/scaricando librerie di predicati.
tuProlog supporta di base la programmazione multi-paradigma, fornendo un modello di integrazione, tra Prolog e i comuni linguaggi di programmazione, pulito e coerente.[8]

Le caratteristiche base di tuProlog

tuProlog è esplicitamente progettato per essere minimale, configurabile dinamicamente e per essere naturalmente integrato con Java e .NET, in modo da supportare la programmazione multi-paradigma/multi-linguaggio in modo semplice e facilmente utilizzabile.

Per minimale si intende che alla base di tutto vi sono gli elementi essenziali di un engine Prolog, quali l'engine di risoluzione e alcuni meccanismi di base ad esso collegati, detti "built-in predicates". Qualsiasi altra funzionalità deve essere implementata tramite librerie (library predicates), oppure tramite nuove teorie caricate e scaricate dinamicamente allo start-up del sistema o a run-time (theory predicates).
La configurabilità è la necessaria controparte della minimalità, permettendo agli utenti di personalizzare il loro sistema tuProlog per i loro bisogni, utilizzando le librerie incluse.
Il facile utilizzo significa che i requisiti per l'installazione sono minimi, e il processo di installazione è semplice come copiare il giusto archivio all'interno della cartella desiderata.
Infine, tuProlog supporta anche l'interoperabilità con i pattern standard di internet (come TCP/IP, RMI, CORBA) e linguaggi e modelli di coordinazione.

Grazie alla sua capacità di supportare la programmazione multi-paradigma, tuProlog permette di:

- Utilizzare qualsiasi classe,libreria o oggetto Java/.NET direttamente dal codice Prolog, supportando il passaggio dei parametri tra le due realtà.
- Utilizzare qualsiasi engine tuProlog direttamente dal codice Java/.NET.
- Migliorare tuProlog definendo nuove librerie, puramente Prolog, object-oriented, o una combinazione fra le due.

- Migliorare Java/.NET con nuovi metodi definiti in modo dichiarativo in Prolog.

2.3 La programmazione logica per l'IoT, utilizzi e vantaggi

Come visto nel capitolo precedente gli scenari dell'Internet of Things richiedono sempre più elementi di intelligenza distribuita.
Questo tipo di richiesta potrebbe essere soddisfatta dalla programmazione logica, e in particolare da Prolog dato che il suo meccanismo di ricerca supporta la valutazione parallela: permettendo di usare più clausole in un dato momento la cui testa coincide con il sotto-obbiettivo corrente (or-parallelism), oppure risolvendo più sotto-obbiettivi simultaneamente, dimostrando multiple associazioni di obbiettivi, se non condividono variabili, allo stesso tempo (and-parallelism).

Le caratteristiche di tuProlog permettono di risolvere molte delle richieste definite dal paradigma dell'Internet of Intelligent Things.
Grazie alla sua minimalità è possibile fornire anche ai dispositivi con risorse limitate un engine logico, fornendo così tale oggetto di una forma di intelligenza utilizzando la possibilità di estendere tale engine con delle librerie ad hoc. Inoltre sfruttando la sua implementazione multi-linguaggio e multi-paradigma è possibile utilizzare questa tecnologia con quasi tutti i linguaggi, tecnologie e standard di comunicazione già in uso nell'ambio dell'IoT.
Si possono quindi utilizzare multiple teorie logiche, associate ai singoli engine e agli oggetti, in grado di descrivere e rappresentare che cosa è vero nell'ambiente d'interesse.

Capitolo 3

Il caso di studio: SmartFridge

3.1 Requisiti

Il sistema deve tenere traccia degli alimenti contenuti all'interno di un frigorifero, e fornire all'utente, il quale utilizza un dispositivo mobile, informazioni riguardanti le ricette realizzabili con il contenuto. Inoltre il sistema deve anche poter informare l'utente di quando sia il momento opportuno di fare la spesa, nel caso gli alimenti in esso contenuti non fossero abbastanza per il numero di persone previste in una data giornata , e compilare una lista adeguata di alimenti da comprare sulla base delle preferenze dell'utente o in base alla loro mancanza.

3.2 Analisi dei requisiti

3.2.1 Casi d'uso

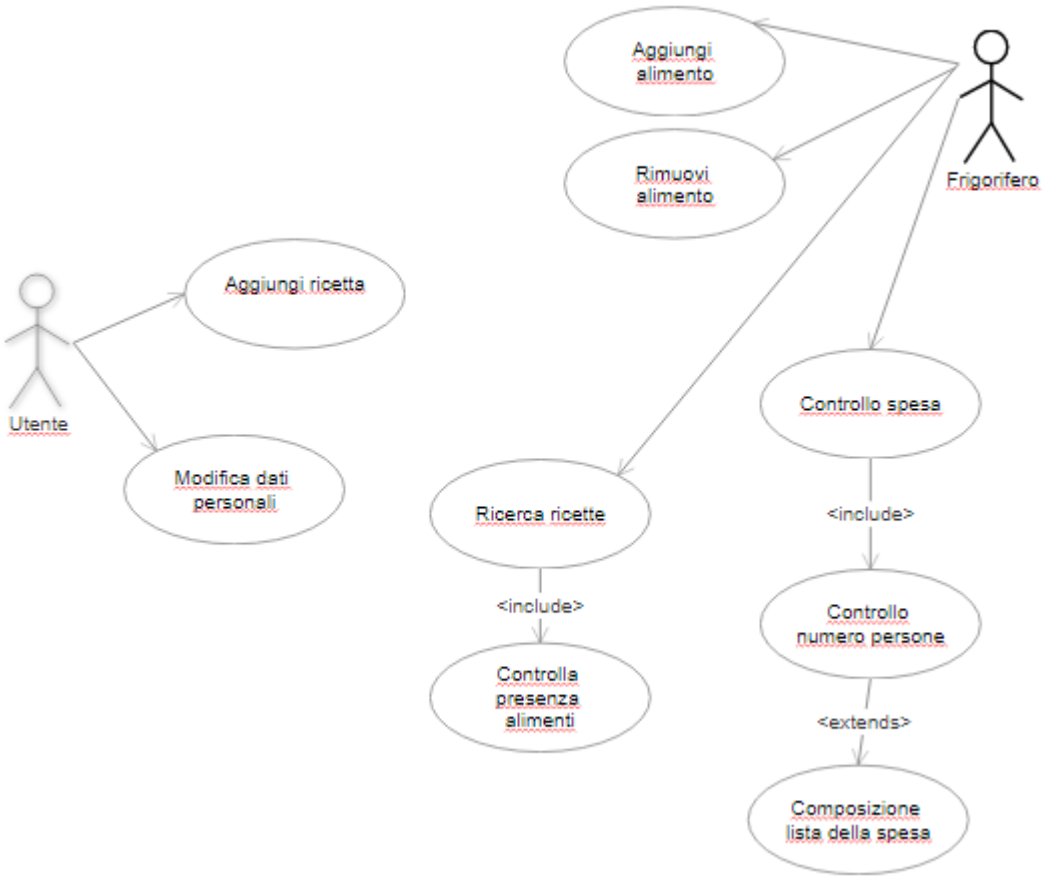

Figura 3.1: Diagramma dei casi d'uso.

3.2.2 Scenari

Nome	Aggiungi Alimento
Descrizione	Il sistema del frigorifero aggiunge un alimento alla base di conoscenza
Attori	Frigorifero
Precondizioni	È stato rilevato l'inserimento di un alimento all'interno del frigorifero
Scenario Principale	I dati riguardanti l'alimento sono aggiunti alla base di conoscenza
Scenario Alternativo	Nel caso tale alimento fosse già presente, ne verrà aggiunta un'unità
Postcondizioni	Il nuovo alimento sarà presente nella base di conoscenza, o sarà presente in quantità maggiori

Tabella 3.1: Scenario di aggiunta di un alimento

Nome	Rimuovi Alimento
Descrizione	Il sistema del frigorifero rimuove un alimento dalla base di conoscenza
Attori	Frigorifero
Precondizioni	È stata rilevata la rimozione di un alimento dall'interno del frigorifero
Scenario Principale	I dati riguardanti l'alimento sono rimossi dalla base di conoscenza
Scenario Alternativo	Nel caso tale alimento fosse presente in più quantità , ne verrà sottratta una sola
Postcondizioni	L'alimento non sarà più presente all'interno della base di conoscenza , o sarà presente in quantità minori

Tabella 3.2: Scenario di rimozione di un alimento

Nome	Aggiungi Ricetta
Descrizione	L'utente aggiunge una ricetta alla base di conoscenza
Attori	Utente e Frigorifero
Precondizioni	
Scenario Principale	L'utente fornisce i dati per la nuova ricetta e il sistema del frigorifero può aggiungerla alla sua base di conoscenza
Scenario Alternativo	L'utente fornisce dei dati errati, quindi il sistema del frigorifero la scarta
Postcondizioni	La nuova ricetta sarà riconosciuta dal sistema del frigorifero

Tabella 3.3: Scenario dell'aggiunta di una nuova ricetta

Nome	Modifica dati personali
Descrizione	L'utente modifica i dati relativi agli orari dei pasti e al numero di persone ad essi presenti
Attori	Utente e Frigorifero
Precondizioni	
Scenario Principale	L'utente fornisce i dati per modificare la base di conoscenza dei pasti, quindi il server può modificare tali entry
Scenario Alternativo	L'utente fornisce dati errati, quindi la base di conoscenza resta invariata
Postcondizioni	L'entry sarà stata modificata, e il sistema del frigorifero si comporterà di conseguenza

Tabella 3.4: Scenario del cambiamento dei dati personali

Nome	Ricerca Ricette
Descrizione	Il sistema del frigorifero controlla quali delle ricette conosciute è possibile fare con gli alimenti presenti al suo interno
Attori	Frigorifero
Precondizioni	Il sistema deve rilevare che si è vicini all'orario di uno dei pasti definiti dall'utente
Scenario Principale	Il sistema interroga la base di conoscenza per verificare quali ricette sia possibile fare e invia i risultati all'utente
Scenario Alternativo	Il sistema interroga la base di conoscenza, ma non trova nessuna ricetta possibile
Postcondizioni	L'utente potrà consultare le possibili ricette

Tabella 3.5: Scenario della ricerca delle ricette disponibili

Nome	Controllo Spesa
Descrizione	Il sistema del frigorifero controlla se è necessario fare la spesa per i giorni seguenti
Attori	Frigorifero
Precondizioni	
Scenario Principale	Il sistema controlla quante persone sono previste durante i pasti del giorno successivo e, in base all'attuale contenuto del frigorifero decide se è necessario effettuare la spesa oppure no
Scenario Alternativo	Nel caso non sia necessario effettuare la spesa, allora il sistema non compilerà una lista per l'utente
Postcondizioni	L'utente riceverà un avviso che lo inviterà a farela spesa, con annessa una lista con degli alimenti consigliati

Tabella 3.6: Scenario del controllo della spesa

3.3 Analisi del problema

La fase di analisi dei requisiti è servita per fissare in modo non ambiguo e formale i requisiti che l'applicazione dovrà soddisfare.

Nella fase di analisi del problema si metterà in luce in che modo saranno realizzate tali specifiche.
In questo sistema per prima cosa sarà necessario analizzare in che modo le due entità remote, l'utente e il frigorifero, possano interagire tra loro. Una possibile soluzione a questo problema è data dallo scambio di informazioni tramite messaggi.
Inoltre in questa fase si terrà particolare attenzione ai vari vantaggi e svantaggi dati dall'utilizzo della programmazione logica per immagazzinare i dati degli alimenti e per permettere al frigorifero di elaborarli per ottenere i risultati richiesti.

3.3.1 Architettura di sistema

La prima idea per una possibile architettura di sistema è data da una semplice architettura client-server, nella quale i sensori raccolgono i dati (nel nostro caso di studio SmartFridge saranno, il nome e, alcune semplici caratteristiche dei nostri prodotti alimentari) che sono poi inviati al server che li immagazzina nella sua Knowledge base.
L'engine logico risiede appunto lato server, ed esso viene interpellato dal sistema per trovare le soluzioni ai problemi discussi nell'analisi dei requisiti. Queste soluzioni sono poi inviate al client, che semplicemente le rappresenta sull'interfaccia utente.

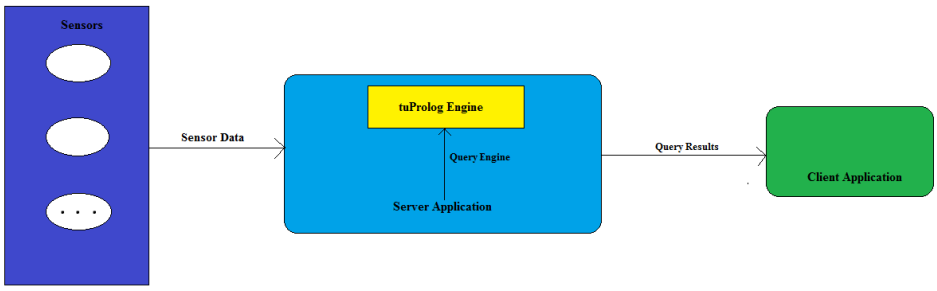

Figura 3.2: Architettura di sistema per l'invio dei dati.

Il client inoltre può configurare certi aspetti ed elementi della Knowledge base , in modo da "informare" il server delle proprie abitudini o particolari avvenimenti, così che il server durante la sua fase di deduzione possa fornire delle soluzioni relative alle esigenze dell'utente, in base alle informazioni che gli sono state fornite, e soprattutto senza che sia l'utente a interpellare il server direttamente per ottenerle.

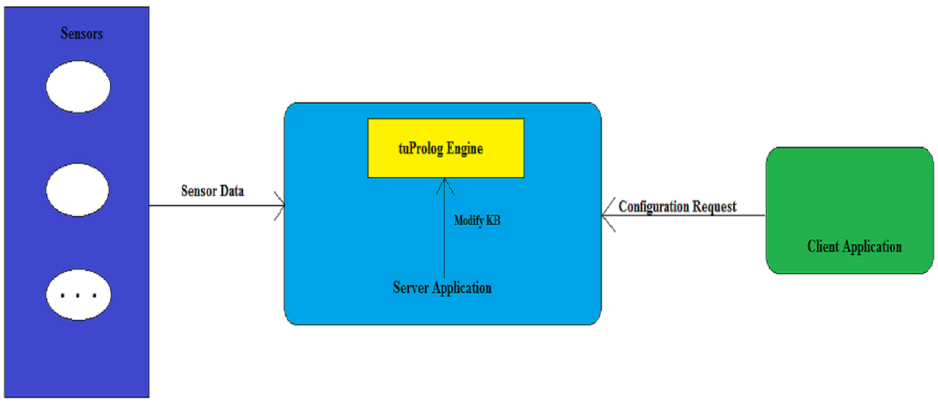

Figura 3.3: Architettura di sistema per la modifica della KB.

Questo tipo di architettura può andare bene per il singolo smart object e il singolo client (come nel caso di studio attuale del frigorifero).

Volendo però possiamo estendere l'architettura per un caso con più oggetti e più client. I singoli componenti rimangono inalterati, risulta però necessario l'utilizzo di un middleware per connettere gli N server con gli eventuali N client (i quali potrebbero risiedere su piattaforme diverse da quelle del nostro caso di studio o anche contenere informazioni diverse).

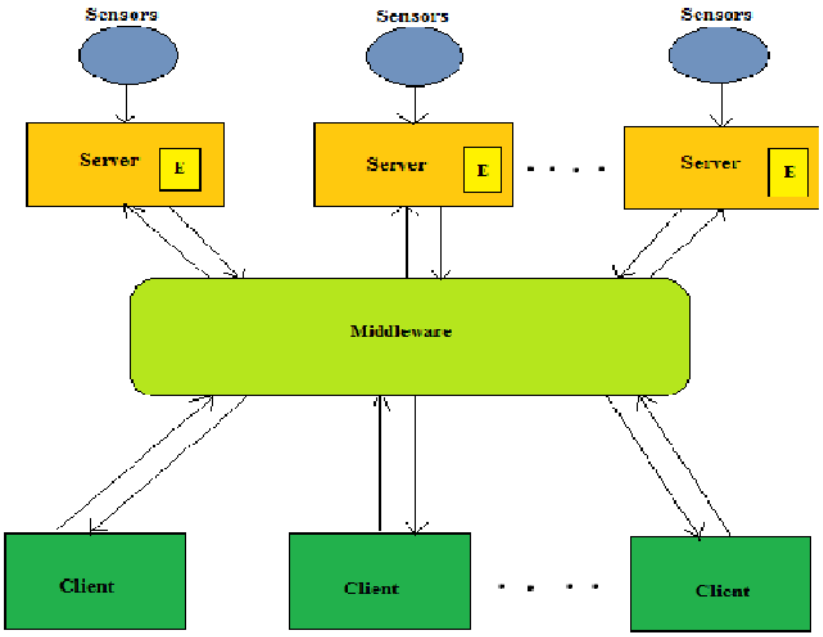

Figura 3.4: Architettura di sistema estesa.

Il client in questo caso potrebbe voler raccogliere soluzioni da due engine diversi, ma legati in un qualche modo (ad esempio frigorifero e varie credenze o cassetti in cucina).Quindi potrebbe essere necessario dotare anche il client di qualche aspetto logico per poter giungere a conclusioni sfruttando i dati di questi engine distinti, modificando leggermente l'approccio a livelli adottato fino ad ora.

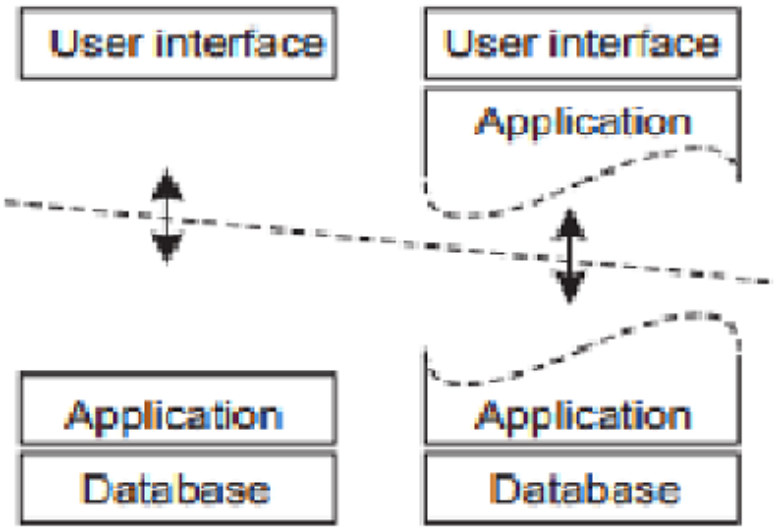

Figura 3.5: Livelli architetturali.

Conseguenze dell'approccio logico per l'architettura

L'approccio logico ha influito minimamente sulla scelta dell'architettura. Infatti le architetture proposte sono per lo più estensioni degli stili classici. L'unica scelta impattante è ricaduta su dove far risiedere l'engine logico, il quale potrebbe stare sia lato server che lato client. La scelta è ricaduta sul lato server in modo da delegare al client la minore quantità di computazione possibile, e per dover inviare una minore mole di dati attraverso il network.

3.3.2 Interazione tra i componenti

Abbiamo visto come il sistema è strutturato descrivendo da quali componenti sarà formato, ora possiamo procedere descrivendo come tali elementi interagiranno fra loro per soddisfare le richieste dei requisiti.

Aggiunta e rimozione di alimenti nella Knowledge Base

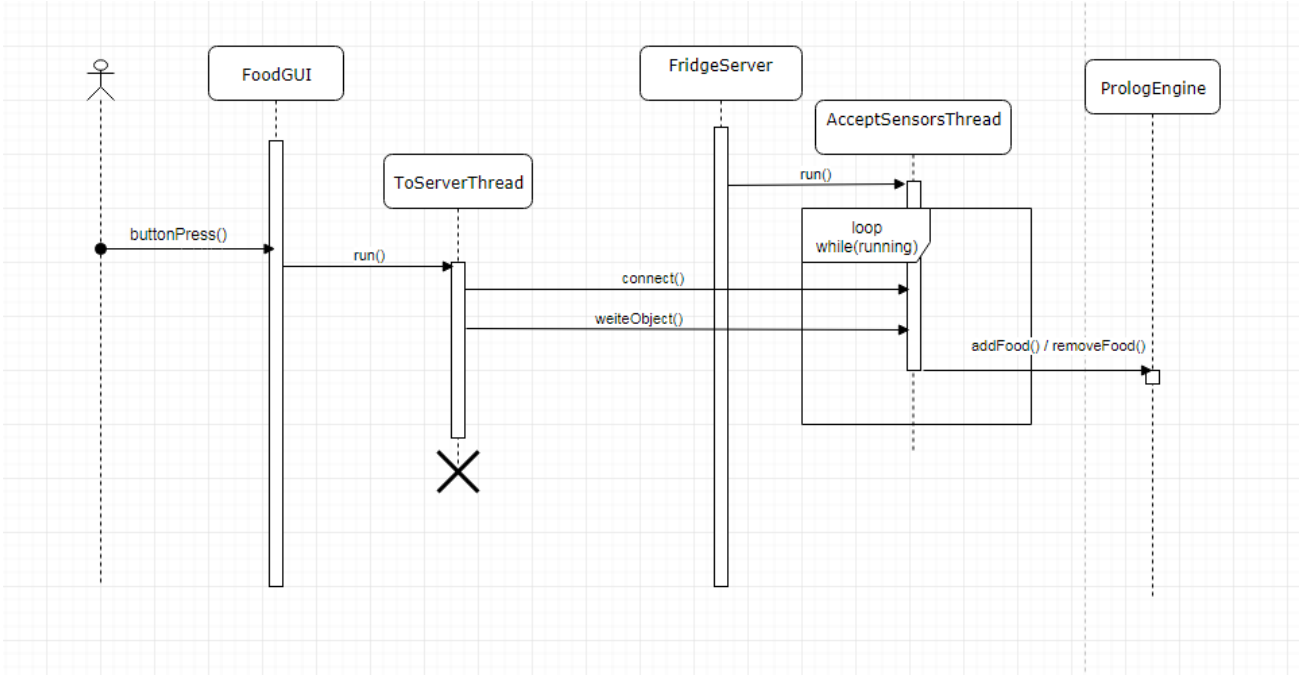

Figura 3.6: Diagramma di sequenza per l'aggiunta e la rimozione degli alimenti.

L'aggiunta e rimozione di un alimento inizia con il rilevamento dell'inserimento o prelievo di un prodotto dall'interno del frigorifero da parte dei sensori ad esso associati. Il sistema dei sensori si occuperà di raccogliere i dati necessari per identificare l'alimento in questione e di inviarli al FridgeServer in modo che possa inserirlo, rimuoverlo o modificarne la quantità all'interno della Knowledge Base di tuProlog.

Il messaggio inviato al FridgeServer, oltre a contenere i dati necessari per identificare l'alimento, avrà anche un flag per definire se l'operazione da effettuare è di aggiunta o di rimozione.
Il sistema FridgeServer avrà sempre attivo un thread che si occupa solamente di rispondere alle richieste di connessione da parte dei sensori e di compiere le richieste, utilizzando i metodi opportuni presenti nella classe PrologEngine, che fornisce l'accesso alle risorse della Knowledge Base.
Per via della mancanza del sistema dei sensori, la creazione dell'evento e la raccolta dei dati sono stati sostituiti da una piccola interfaccia grafica in Java, dove la pressione di un JButton segnala l'evento di aggiunta o rimozione.

Richiesta di un servizio da parte dell'utente.

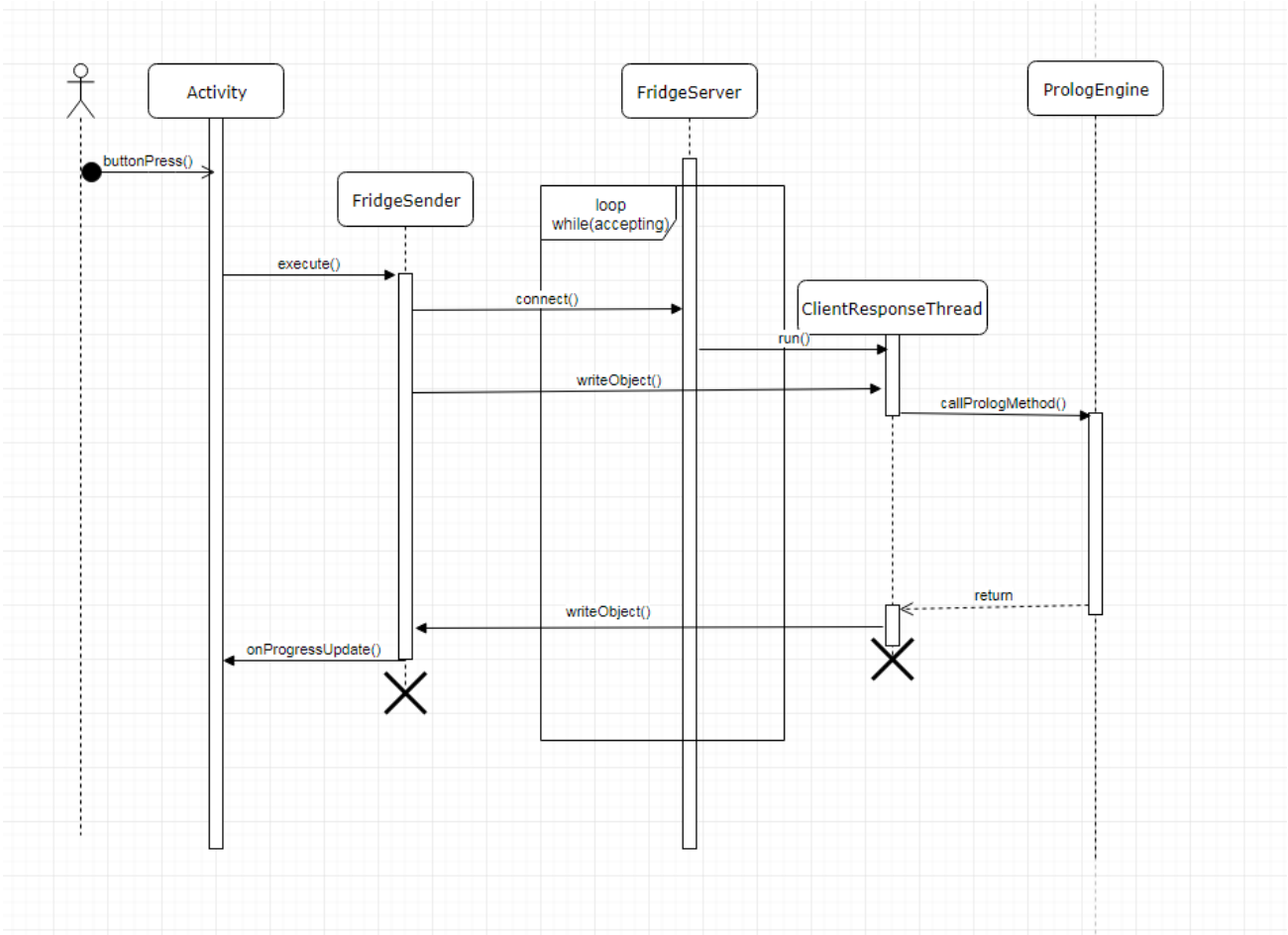

Figura 3.7: Diagramma di sequenza per la gestione delle richieste dell'utente.

L'utente può interagire con il frigorifero inviando richieste specifiche, per sfruttare le funzionalità dell'engine tuProlog, tramite l'interfaccia di un'applicazione Android.
Alla pressione di determinati Button e inserendo i dati appropriati, l'utente è in grado di richiedere al sistema tutte le operazioni definite dai requisiti.

La richiesta avviene sempre nello stesso modo: l'applicazione Android, dopo la pressione del pulsante, crea un Task che compone un oggetto contenente

tutti i dati necessari, più un flag per identificare il tipo di operazione richiesta; il sistema FridgeServer sarà sempre in ascolto per delle connessioni e una volta ricevuta creerà un thread apposito per gestire la richiesta e rispondere all'utente con i dati opportuni.

Si è deciso di creare un thread apposito per gestire le risposte in modo che il sistema possa continuare ad attendere altre connessioni da parte di altri utenti.

Ricezione passiva dei dati da parte dell'utente.

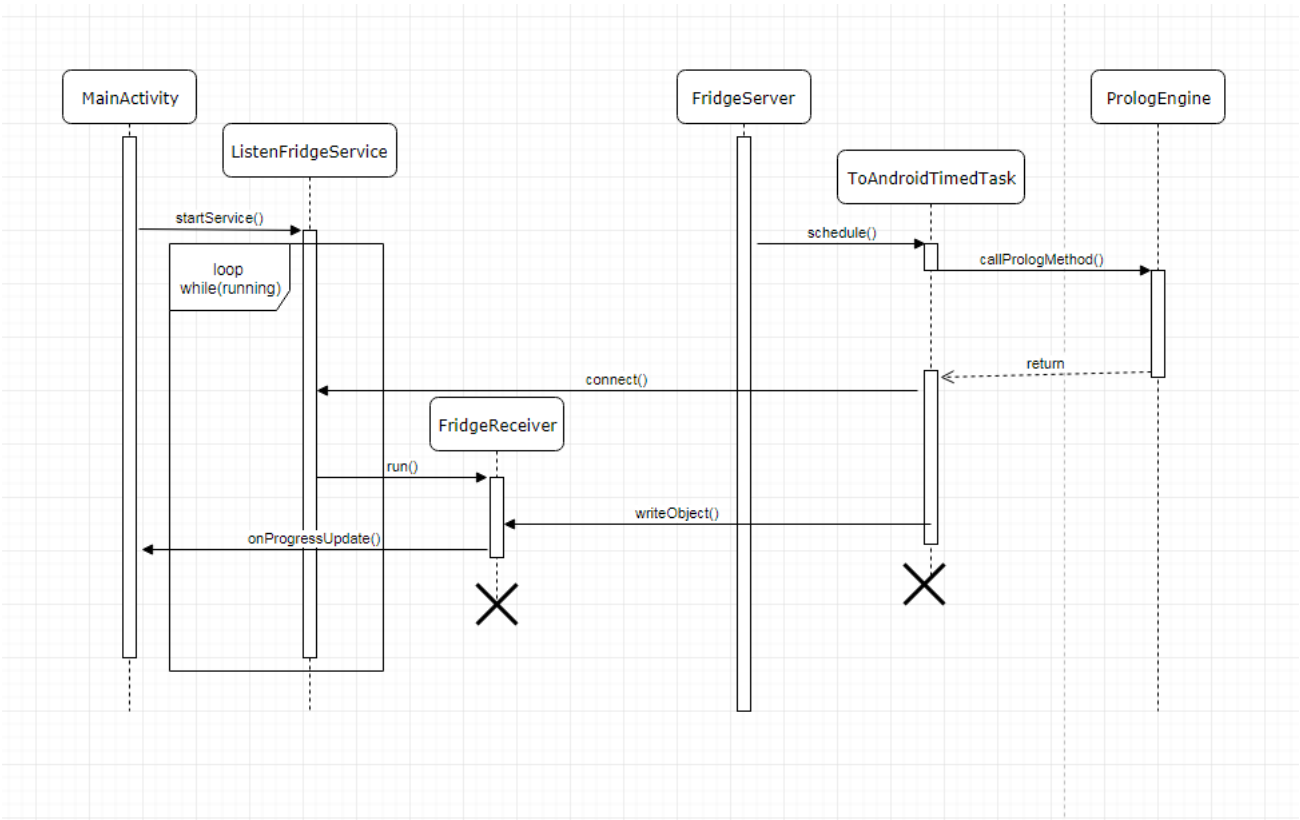

Figura 3.8: Diagramma di sequenza per l'invio programmato dei dati del server.

Infine l'utente può anche fornire i propri dati al sistema FridgeServer in modo da registrarsi per ottenere, ad un orario preciso definito dall'utente, le funzionalità di tuProlog in modo passivo.

Il sistema FridgeServer, al suo avvio, programmerà un Task per essere eseguito ad un orario preciso, questo Task si occuperà di mandare a tutti i client registrati la lista delle ricette per il prossimo pasto, e infine, prelevando l'informazione dalla Knowledge Base programmerà un nuovo Task per il pasto successivo a quello appena controllato.
L'applicazione Android rimarrà sempre in attesa di una connessione tramite un servizio in background, il quale al momento dell'esecuzione del Task creerà un thread apposito per la ricezione della lista di ricette e, se l'applicazione si trova in background oppure non è avviata, tale lista rimarrà salvata e disponibile fino al prossimo avvio.
In questo modo l'utente potrà ricevere i dati da parte del frigorifero senza che egli debba richiederli manualmente.

3.4 Progetto e implementazione

Dopo aver descritto i modi in cui gli elementi interagiscono, si può passare a descrivere come sono implementati gli elementi più importanti del sistema.

3.4.1 La Knowledge Base

La base di conoscenza dell'engine tuProlog è composta da tre file di testo.

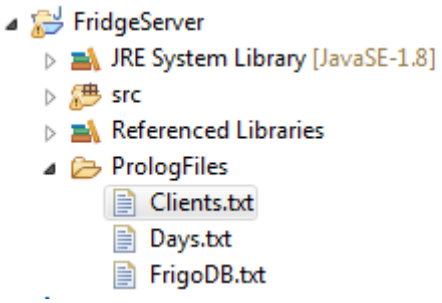

Figura 3.9: Organizzazione Knowledge Base.

Nel file "FrigoDB" saranno immagazzinati i contenuti attuali del frigorifero, assieme alle ricette che avrà a disposizione.
Inoltre sono presenti alcune operazioni per il conteggio degli elementi e una lista di alimenti preferiti e mancanti utilizzati per controllare se sia necessario o meno fare la spesa.

Gli alimenti presenti all'interno del frigorifero sono rappresentati da un "fact" contenete il loro nome (utilizzato principalmente come identificatore dell'alimento), il tipo di alimento, la categoria a cui fanno parte e la quantità presente al momento nel frigorifero (con 1 la quantità minima per essere presente), ad esempio:

```
food('Petto di Pollo','Pollo','Carne',2).
food('Carote Sfuse','Carote','Verdura',5).
food('Mele Fuji','Mele','Frutta',3).
food('Cioccolata extra fondende','Cioccolata','Dolci',2).
food('Prosciutto Crudo','Prosciutto','Carne',2).
food('Pomodori Datterino','Pomodori','Verdura',4).
food('Latte intero','Latte','Bevande',1).
food('Latte Parzialmente Scremato','Latte','Bevande',1).
food('Insalata Iceberg','Insalata','Verdura',2).
food('Mais Dolce','Mais','Verdura',3).
food('Fagioli Borlotti','Fagioli','Verdura',2).
food('Uova','Uova','Uova',4).
```

Le ricette sono invece modellate come "rules" contenenti il loro nome, il pasto per cui potrebbero essere cucinate e una lista di ingredienti necessari per far si che si possano fare, infatti per far si che una ricetta risulti all'interno della soluzione di una query, è necessario che gli ingredienti necessari facciano parte del set di "fact" degli alimenti contenuti all'interno del frigorifero.

Le ricette sono pensate in modo che se sono presenti più alimenti con lo stesso "Tipo" e "Categoria", ma con identificatori diversi, durante la loro ricerca possano apparire più volte, ma con ingredienti diversi, poichè facendo parte delle stesse categorie tali ingredienti possono essere intercambiabili e quindi si lascia all'utente la possibilità di scegliere quale combinazione di alimenti usare per la creazione della ricetta.
Alcuni esempi:

```
recipe('Pollo con cipolle','Pranzo',[ID1,ID2]) :-
    food(ID1,'Pollo','Carne',N1),food(ID2,'Cipolle','Verdura',N2).
recipe('Pollo con patate','Cena',[ID1,ID2]) :-
```

```
    food(ID1,'Pollo','Carne',_),food(ID2,'Patate','Verdura',_).
recipe('Insalata mista','Pranzo',[ID1,ID2,ID3,ID4]) :-
    food(ID1,'Insalata','Verdura',_),food(ID2,'Mais','Verdura',_),
   food(ID3,'Carote','Verdura',_),food(ID4,'Pomodori','Verdura',_).
recipe('Uova Strapazzate','Cena',[ID1]) :- food(ID1,'Uova','Uova',_).
recipe('Uova Strapazzate','Pranzo',[ID1]) :- food(ID1,'Uova','Uova',_).
recipe('Latte e biscotti','Colazione',[ID1,ID2]) :-
    food(ID1,'Latte','Bevande',_),food(ID2,'Biscotti','Dolci',_).
```

Alcune ricette sono presenti a priori, altre saranno inserite dall'utente in modo manuale.

Infine per aiutare nella composizione di una lista della spesa, all'interno della Knowledge Base sono presenti dei "fact" che rappresentano alimenti che erano presenti in precedenza nel frigorifero, ma che sono stati prelevati (che fanno parte quindi di possibili alimenti da comprare nuovamente in futuro) e delle Regole che rappresentano degli alimenti essenziali, cioè che non dovrebbero mai mancare all'interno del frigorifero, infatti se si effettua una query, essi risultano soltanto se l'alimento da loro descritto non è presente all'interno del frigorifero.
Questo accade sfruttando il meccanismo del "not" di Prolog.

```
essentialfood('Mele','Frutta') :- \+ food(_,'Mele','Frutta',_).
essentialfood('Uova','Uova') :- \+ food(_,'Uova','Uova',_).
essentialfood('Burro','Formaggio') :- \+ food(_,'Burro','Formaggio',_).
essentialfood('Parmigiano','Formaggio') :- \+
    food(_,'Parmigiano','Formaggio',_).
essentialfood('Carote','Verdura') :- \+ food(_,'Carote','Verdura',_).
essentialfood('Insalata','Verdura') :- \+ food(_,'Insalata','Verdura',_).
essentialfood('Limoni','Frutta') :- \+ food(_,'Limoni','Frutta',_).
essentialfood('Prosciutto','Carne') :- \+ food(_,'Prosciutto','Carne',_).
essentialfood('Latte','Bevande') :- \+ food(_,'Latte','Bevande',_).
```

```
foodtobuy('Bacon','Maiale','Carne').
foodtobuy('Alette Di Pollo','Pollo','Carne').
foodtobuy('Miele','Miele','Dolci').
foodtobuy('Burrata','Burrata','Formaggio').
foodtobuy('Zucchine','Zucchine','Verdura').
```

All'interno del secondo file "Days" è presente un insieme di "facts" che rappresentano tutti i pasti presenti all'interno di ogni giorno della settimana, definendo l'orario in cui avvengono e il numero di persone partecipanti.
Questo file è usato per tenere traccia dell'orario dei pasti e del numero di persone che ci si aspetta un determinato giorno, così da pianificare la necessità di fare la spesa e per fornire le ricette adeguate in base agli orari.

```
meal('Pranzo','MONDAY','12:30','2').
meal('Cena','MONDAY','19:30','4').
meal('Colazione','MONDAY','7:30','3').

meal('Pranzo','TUESDAY','12:30','1').
meal('Cena','TUESDAY','19:30','4').
meal('Colazione','TUESDAY','7:30','3').

meal('Pranzo','WEDNESDAY','12:30','3').
meal('Cena','WEDNESDAY','19:30','4').
meal('Colazione','WEDNESDAY','7:30','3').

meal('Pranzo','THURSDAY','12:30','2').
meal('Cena','THURSDAY','19:30','4').
meal('Colazione','THURSDAY','7:30','3').

meal('Pranzo','FRIDAY','12:30','2').
meal('Cena','FRIDAY','19:30','3').
meal('Colazione','FRIDAY','7:30','3').
```

```
meal('Pranzo','SATURDAY','13:00','4').
meal('Cena','SATURDAY','19:30','3').
meal('Colazione', 'SATURDAY','9:30','3').

meal('Pranzo','SUNDAY','13:00','4').
meal('Cena','SUNDAY','19:30','4').
meal('Colazione', 'SUNDAY','9:30','3').
```

Infine all'interno dell'ultimo file di testo "Clients" saranno presenti i client registrati, che saranno selezionati per ricevere le ricette in modo passivo agli orari programmati all'interno del file "Days".

```
client ('ID0','192.168.1.102').
client ('ID1','192.168.1.112').
client ('ID2','192.168.1.106').
```

L'accesso alle risorse dell'engine tuProlog sarà regolato da due Lock, uno per gestire l'accesso ai file riguardanti i dati dei giorni e il contenuto del frigorifero, e l'altro per l'accesso al file relativo ai dati dei client.
Questa misura è stata necessaria per impedire delle inconsistenze dovute alle possibili modifiche e richieste effettuate da multipli client contemporaneamente.
Grazie a questi Lock, ogni thread che gestisce la connessione con i client qualsiasi volta che deve accedere ad una risorsa chiama il metodo Lock() sincronizzato, in modo tale che gli altri thread attendano fino al completamento delle operazioni del thread che ha ottenuto l'accesso, prima di procedere con le loro istruzioni.

3.4.2 I metodi per l'utilizzo dell'engine logico, e l'interfaccia Android

In questa sezione verranno descritti i vari metodi utilizzati per interrogare l'engine logico tuProlog, mostrando il loro funzionamento e come i risultati vengono mostrati sull'applicazione android dell'utente.

Il sistema FridgeServer

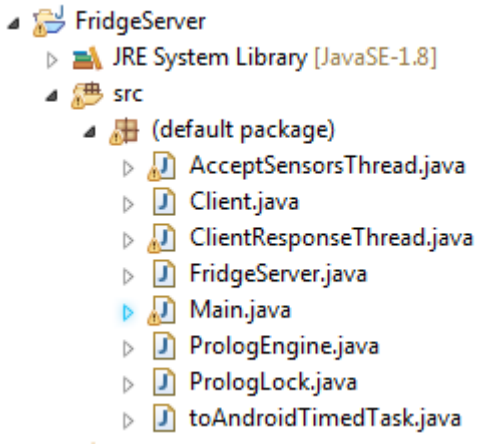

Figura 3.10: Organizzazione sistema FridgeServer.

Il sistema implementa l'interazione con l'engine tuProlog all'interno della classe "PrologEngine", la quale contiene l'engine tuProlog e tutti i metodi usati per sfruttare le sue potenzialità.

L'applicazione Android

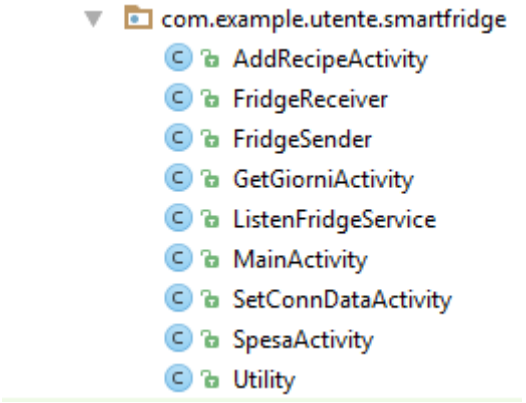

Figura 3.11: Organizzazione applicazione Android.

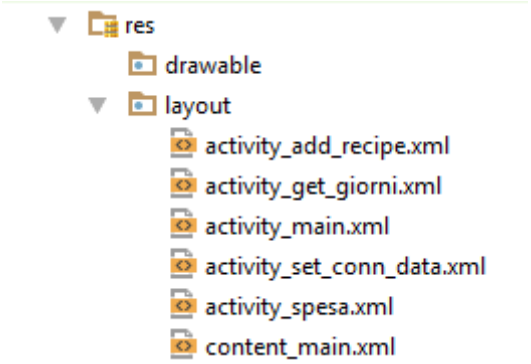

Figura 3.12: File di layout Android.

Aggiunta e rimozione degli alimenti

Il sistema rimane in ascolto per i dati ricevuti dai sensori tramite un AcceptSensorsThread, il quale una volta ricevuto l'oggetto contenente i dati necessari chiama il metodo addFood() o removeFood() della classe PrologEngine, a seconda del flag ricevuto.

```
public class AcceptSensorsThread extends Thread {
```

```
protected PrologLock prologLock;
protected PrologEngine prologMethods;
protected ObjectInputStream in;
protected List<String> received;
protected boolean running;

public AcceptSensorsThread(PrologLock prologLock,PrologEngine
    prologMethods) {
   this.prologLock = prologLock;
   this.prologMethods = prologMethods;
   this.running = true;
}

public void run() {
   try {
      ServerSocket accepter = new ServerSocket(9997);
      while(running) {
         Socket socket = accepter.accept();
         in = new ObjectInputStream(socket.getInputStream());
         received = (List<String>) in.readObject();
         //Il flag che indica l'aggiunta o la rimozione
         String isAdd = received.get(0);
         //I tre dati necessari per identificare un alimento
         String foodReceived = received.get(1);
         String catReceived = received.get(2);
         String cat2Received = received.get(3);
         prologLock.lock();
         if(isAdd.equals("ADD")) {
            prologMethods.addFood(foodReceived, catReceived,
                cat2Received);
         }else if(isAdd.equals("REMOVE")) {
            prologMethods.removeFood(foodReceived, catReceived,
                cat2Received);
```

```
                }
                prologLock.unlock();
                in.close();
                socket.close();
            }
            accepter.close();

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

}
```

I due metodi si occupano prima di tutto di identificare se l'alimento in questione è già presente all'interno del frigorifero, effettuando una query.
Nel caso di addFood(), se l'alimento è già presente, sarà aumentata la sua quantità, mentre se l'alimento non è presente, sarà inserito all'interno del file con la struttura vista in precedenza, e se quest'ultimo era presente negli alimenti da comprare in futuro verrà rimosso da quella categoria.
Nel caso di removeFood() se l'alimento è presente, la sua quantità sarà ridotta, e se raggiungerà il valore zero la sua occorrenza sarà rimossa dalla Knowledge Base ed inserita negli alimenti che possono essere acquistati in futuro.

```
public void addFood(String foodReceived,String catReceived,String
    cat2Received) {
      SolveInfo info;
      String food =
          "food('"+foodReceived+"','"+catReceived+"','"+cat2Received+"',X).";
      String pastFood =
          "foodtobuy('"+foodReceived+"','"+catReceived+"','"+cat2Received+").";
      try {
```

```
            Theory nTheory = new Theory(new FileInputStream(frigoDBPath));
            engine.setTheory(nTheory);
            info = engine.solve(food);
            if (!(info.isSuccess())) {
                String newLine =
                    "food('"+foodReceived+"','"+catReceived+"','"+cat2Received+"',1).";
                addFoodLine(newLine); //Aggiunta nuovo alimento
                SolveInfo existingInfo = engine.solve(pastFood);
                if(existingInfo.isSuccess()) {
                    alterLine(pastFood,null,frigoDBPath,true);//Rimozione dagli
                        alimenti da comprare
                }

            }else if(info.isSuccess()){
                engine.clearTheory();
                int count = Integer.parseInt(info.getVarValue("X").toString());
                String oldLine = "food('"+foodReceived+"',
                    '"+catReceived+"','"+cat2Received+"',"+count+").";
                count++;
                String newLine = "food('"+foodReceived+"',
                    '"+catReceived+"','"+cat2Received+"',"+count+").";
                alterLine(oldLine,newLine,frigoDBPath,false);//Modifica il numero
                    dell'alimento
            }
            nTheory = new Theory(new FileInputStream(frigoDBPath));
            engine.setTheory(nTheory);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    public void removeFood(String foodReceived,String catReceived,String
        cat2Received) {
```

```
String foodToFind =
    "food('"+foodReceived+"','"+catReceived+"','"+cat2Received+"',X).";
try {
    Theory nTheory;
    nTheory = new Theory(new FileInputStream(frigoDBPath));
    engine.setTheory(nTheory);
    SolveInfo info = engine.solve(foodToFind);//Controllo presenza
        alimento
    if(info.isSuccess()) {
        String xValue = info.getVarValue("X").toString();
        int xInt = Integer.parseInt(xValue);
        String foodToDecrease ="food('"+foodReceived+"',
            '"+catReceived+"','"+cat2Received+"',"+xValue+").";
        if((xInt-1) <= 0) {
            alterLine(foodToDecrease,null,frigoDBPath,true);
                //Eliminazione occorrenza alimento
            String pastFood = "foodtobuy('"+foodReceived+"',
                '"+catReceived+"','"+cat2Received+"').";
            addFoodLine(pastFood); //Aggiunta agli alimenti da comprare

        }else {
            xInt--;
            String foodDecreased ="food('"+foodReceived+"',
                '"+catReceived+"','"+cat2Received+"',"+xInt+").";
            alterLine(foodToDecrease,foodDecreased,frigoDBPath,false);//Modifica
                il numero dell'alimento
        }
    }
    nTheory = new Theory(new FileInputStream(frigoDBPath));
    engine.setTheory(nTheory);
}catch (Exception e) {
    e.printStackTrace();
}
```

```
}
```

Ricerca delle ricette

Il meccanismo di ricerca delle ricette inizia con la richiesta da parte dell'utente oppure dal raggiungimento dell'orario con cui si è programmato il Task per l'invio ai client registrati.
Una volta chiamato, il metodo searchRecipes() inizia raccogliendo l'orario in quel preciso istante e gli orari dei pasti dello stesso giorno presenti all'interno della Knowledge Base, dopodichè continua controllando in quale fascia oraria ci si trova e a seconda del risultato decide se si dovrà cercare ricette per colazione pranzo o cena effettuando la query di conseguenza e popolando una lista con tutte le soluzioni trovate, che verrà poi inviata all'utente in risposta alla sua richiesta, oppure inviata a tutti i client registrati.

```
public ArrayList<String> searchRecipes(){

    ArrayList<String> list = new ArrayList<String>();
    ZonedDateTime time = ZonedDateTime.now(); //Otteniamo l'ora attuale
    String now = ""+time.getHour()+":"+time.getMinute();
    String [] timeu = now.split ( ":" );
    int nowHour = Integer.parseInt ( timeu[0].trim());
    engine.clearTheory();
    Theory nTheory;
    try {
        nTheory = new Theory(new FileInputStream(daysPath));
        engine.setTheory(nTheory);
        //Otteniamo l'ora di colazione
        SolveInfo info = engine.solve("meal('Colazione',
                '"+time.getDayOfWeek().toString()+"',B,C).");
        String orarioCol = info.getVarValue("B").toString();
        orarioCol = orarioCol.substring(1, orarioCol.length()-1);
        timeu = orarioCol.split ( ":" );
```

```
int colHour = Integer.parseInt (timeu[0].trim());
//Otteniamo l'ora di pranzo
info = engine.solve("meal('Pranzo',
        '"+time.getDayOfWeek().toString()+"',B,C).");
String orarioPra = info.getVarValue("B").toString();
orarioPra = orarioPra.substring(1, orarioPra.length()-1);
timeu = orarioPra.split ( ":" );
int praHour = Integer.parseInt ( timeu[0].trim());
//Otteniamo l'ora di cena
info = engine.solve("meal('Cena',
        '"+time.getDayOfWeek().toString()+"',B,C).");
String orarioCen = info.getVarValue("B").toString();
orarioCen = orarioCen.substring(1, orarioCen.length()-1);
timeu = orarioCen.split ( ":" );
int cenHour = Integer.parseInt ( timeu[0].trim() );

 nTheory = new Theory(new FileInputStream(frigoDBPath));
 engine.setTheory(nTheory);
if(nowHour > cenHour) {
    //Mostra le ricette per la colazione
    info = engine.solve("recipe(Nome,'Colazione',Ingredienti).");
    while (info.isSuccess()){
        list.add("Nome "+info.getVarValue("Nome")+"
            Ingredienti:"+info.getVarValue("Ingredienti"));
        if (engine.hasOpenAlternatives()){
            info=engine.solveNext();
        } else {
            break;
        }
    }
}else if(nowHour <= praHour && nowHour > colHour) {
    //Mostra le ricette per il pranzo
    info = engine.solve("recipe(Nome,'Pranzo',Ingredienti).");
```

```
            while (info.isSuccess()){
                list.add("Nome "+info.getVarValue("Nome")+"
                    Ingredienti:"+info.getVarValue("Ingredienti"));
                if (engine.hasOpenAlternatives()){
                    info=engine.solveNext();
                } else {
                    break;
                }
            }
        }else if(nowHour > praHour && nowHour <= cenHour) {
            //Mostra le ricette per la cena
            info = engine.solve("recipe(Nome,'Cena',Ingredienti).");
            while (info.isSuccess()){
                list.add("Nome "+info.getVarValue("Nome")+"
                    Ingredienti:"+info.getVarValue("Ingredienti"));
                if (engine.hasOpenAlternatives()){
                    info=engine.solveNext();
                } else {
                    break;
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return list;
}
```

L'applicazione android poi, una volta ricevuta la lista delle ricette, procede a mostrarla nella sua activity principale.
Questa activity principale contiene anche una serie di bottoni e un menu.
Il bottone "Ricette" si occupa di mandare al server una richiesta per ottenere le ricette per il prossimo pasto, mentre i bottoni "Spesa", "Giorni" e "Ag-

giungi Ricetta", assieme al menu, portano tutti a nuove activity utilizzate per richiedere e mostrare i risultati di tutte le altre funzionalità.

Figura 3.13: Activity principale applicazione android.

Controllo per la spesa e compilazione della lista

Per controllare se si deve fare la spesa, la prima operazione che sarà effettuata è quella di controllare se all'interno del frigorifero sono presenti gli alimenti essenziali, specificati nella Knowledge Base, effettuando l'opportuna query all'engine tuProlog, e il sistema inserirà nella lista tutti gli alimenti essenziali mancanti.
Dopo aver controllato gli alimenti essenziali, inizierà un meccanismo che a seconda dell'orario in cui arriva la richiesta, calcola il numero di persone che ci si aspetta per i pasti rimanenti di quel giorno, e mette in relazione il numero trovato con la quantità di alimenti di ogni possibile categoria presenti nel frigorifero e, ogni volta che il numero di persone risulta essere maggiore del numero di alimenti presenti, la categoria di cui fanno parte verrà etichettata

come "lacking" (carente) e quindi sarà un possibile candidato per la lista della spesa che verrà compilata in seguito.
Nel caso non ci sia bisogno di fare la spesa per lo stesso giorno in cui arriva la richiesta, il sistema eseguirà comunque un controllo anche per il giorno seguente, tenendo conto del numero di persone ottenuto al passo precedente, procedendo allo stesso modo descritto in precedenza.
Nel caso siano trovate delle categorie di alimenti "lacking" il sistema procederà con il popolamento di una lista con dei possibili alimenti comprati in passato facenti parte di ogni categoria "lacking" trovate, tale lista sarà poi concatenata con quella degli alimenti essenziali mancanti nel caso ve ne fossero.

```
public ArrayList<String> checkMissingEssentials() {
    ArrayList<String> essentialsList = new ArrayList<String>();
     essentialsList .add("Alimenti essenziali");
    Theory nTheory;
    try {
        nTheory = new Theory(new FileInputStream(frigoDBPath));
        engine.setTheory(nTheory);
        //Query per il controllo degli essenziali mancanti
        SolveInfo info = engine.solve("essentialfood(X,Y).");
        while (info.isSuccess()){
            String missingEssential = info.getVarValue("X").toString();
             essentialsList .add(missingEssential);
             if (engine.hasOpenAlternatives()){
                info=engine.solveNext();
            } else {
                break;
            }
        }
    }catch(Exception e) {
        e.printStackTrace();
```

```
        }
        return essentialsList ;
    }
```

```
public boolean checkSpesa() {
        boolean spesaIsTrue = false;
        this.lackingCategories.clear();
        Theory nTheory;
        try {
            engine.clearTheory();
            nTheory = new Theory(new FileInputStream(daysPath));
            engine.setTheory(nTheory);
            ZonedDateTime time = ZonedDateTime.now();//Otteniamo l'orario
                attuale
            String now = ""+time.getHour()+":"+time.getMinute();String[]
                timeu = now.split ( ":" );
            int nowHour = Integer.parseInt ( timeu[0].trim());

            //Otteniamo l'orario della colazione
            SolveInfo info =
                engine.solve("meal('Colazione','"+time.getDayOfWeek().toString()+"',B,C).");
            String orarioCol = info.getVarValue("B").toString();
            orarioCol = orarioCol.substring(1, orarioCol.length()-1);
            timeu = orarioCol.split ( ":" );
            int colHour = Integer.parseInt ( timeu[0].trim() );
            String p;

            //Otteniamo l'orario del pranzo e il numero di persone previsto
            info =
                engine.solve("meal('Pranzo','"+time.getDayOfWeek().toString()+"',B,C).");
            String orarioPra = info.getVarValue("B").toString();
            orarioPra = orarioPra.substring(1, orarioPra.length()-1);
            timeu = orarioPra.split ( ":" );
            int praHour = Integer.parseInt ( timeu[0].trim());
```

```
p = info.getVarValue("C").toString();
p = p.substring(1,p.length()-1);
int pPranzo = Integer.parseInt(p);

//Otteniamo l'orario della cena e il numero di persone previsto
info =
    engine.solve("meal('Cena',"+time.getDayOfWeek().toString()+",B,C).");
String orarioCen = info.getVarValue("B").toString();
orarioCen = orarioCen.substring(1, orarioCen.length()-1);
timeu = orarioCen.split ( ":" );
int cenHour = Integer.parseInt ( timeu[0].trim() );
p = info.getVarValue("C").toString();
p = p.substring(1,p.length()-1);
int pCena = Integer.parseInt(p);

engine.clearTheory();
nTheory = new Theory(new FileInputStream(frigoDBPath));
engine.setTheory(nTheory);

if(nowHour > cenHour) {
    //Se il prossimo pasto e' la colazione di domani , allora si
        procede con il controllo per il giorno dopo
    String tomorrow = getNextDay(time.getDayOfWeek().toString());
    spesaIsTrue = checkForTomorrow(tomorrow,0);
}else if(nowHour <= praHour && nowHour > colHour) {
    //Se il prossimo pasto e' il pranzo , si controlla se si ha
        abbastanza cibo per pranzo e cena
    int pForToday = pPranzo+pCena;
    for(int i=0;i<categoryList.length;i++) {
        info = engine.solve("countfood(S,"+categoryList[i]+").");
        int categoryCount =
            Integer.parseInt(info.getVarValue("S").toString());
            if(pForToday > categoryCount) {
```

```
                    spesaIsTrue = true;
                    lackingCategories.add(categoryList[i]);
                }
            }
            if (!spesaIsTrue) {
            String tomorrow = getNextDay(time.getDayOfWeek().toString());
            spesaIsTrue = checkForTomorrow(tomorrow,pForToday);
            }

        }else if (nowHour > praHour && nowHour <= cenHour) {
            //Se il prossimo pasto e' la cena, si controlla solo per il
                numero di persone atteso per essa
            for(int i=0;i<categoryList.length;i++) {
                info = engine.solve("countfood(S,'"+categoryList[i]+"').");
                int categoryCount =
                    Integer.parseInt(info.getVarValue("S").toString());
                    if(pCena > categoryCount) {
                        spesaIsTrue = true;
                        lackingCategories.add(categoryList[i]);
                    }
                }
            if (!spesaIsTrue) {
            String tomorrow = getNextDay(time.getDayOfWeek().toString());
            spesaIsTrue = checkForTomorrow(tomorrow,pCena);
            }
        }
    }catch(Exception e) {
        e.printStackTrace();
    }
    return spesaIsTrue;
}
```

```
public boolean checkForTomorrow(String day,int pForYesterday) {
    boolean spesaTomorrow = false;
```

```
        Theory nTheory;
        int pGiorno;
        try {
            nTheory = new Theory(new FileInputStream(daysPath));
            engine.setTheory(nTheory);
            //Otteniamo il numero di persone atteso per l'intera giornata
            SolveInfo info = engine.solve("meal('Colazione','"+day+"',B,C).");
            pGiorno = Integer.parseInt(info.getVarValue("C").toString());
            info = engine.solve("meal('Pranzo','"+day+"',B,C).");
            pGiorno = +Integer.parseInt(info.getVarValue("C").toString());
            info = engine.solve("meal('Cena','"+day+"',B,C).");
            pGiorno = +Integer.parseInt(info.getVarValue("C").toString());
            int pTot = pGiorno+pForYesterday;

            for(int i=0;i<categoryList.length;i++) {
                info = engine.solve("countfood(S,'"+categoryList[i]+"').");
                int categoryCount =
                    Integer.parseInt(info.getVarValue("S").toString());
                if(pTot > categoryCount) {
                    spesaTomorrow = true;
                    lackingCategories.add(categoryList[i]);
                }
            }
            nTheory = new Theory(new FileInputStream(frigoDBPath));
            engine.setTheory(nTheory);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return spesaTomorrow;
    }
```

```
public ArrayList<String> getSpesaList(){
        ArrayList<String> spesaList = new ArrayList<String>();
        SolveInfo info;
```

```
    try {
       if(checkSpesa()) {
          //Componiamo la lista in base alle categoria trovate
              "carenti" ai passi precedenti
          for(int i=0;i<this.lackingCategories.size();i++) {
             info =
                 engine.solve("foodtobuy(X,Y,'"+lackingCategories.get(i)+"').");
             spesaList.add("Categoria :"+lackingCategories.get(i));
             while(info.isSuccess()) {
                String element = info.getVarValue("X").toString();
                spesaList.add(element);
                if(info.hasOpenAlternatives()) {
                   engine.solveNext();
                }else {
                   break;
                }
             }
          }
       }
    }catch(Exception e) {
       e.printStackTrace();
    }
    return spesaList;
 }
```

Tale lista sarà poi inviata all'utente che ha inviato la richiesta , che la mostrerà all'interno di una lista sull'interfaccia dell'applicazione Android.
L'activity in questione conterrà un bottone "Controllo Spesa" utilizzata per richiedere un controllo da parte del server, e una lista utilizzata per mostrare il risultato ottenuto.

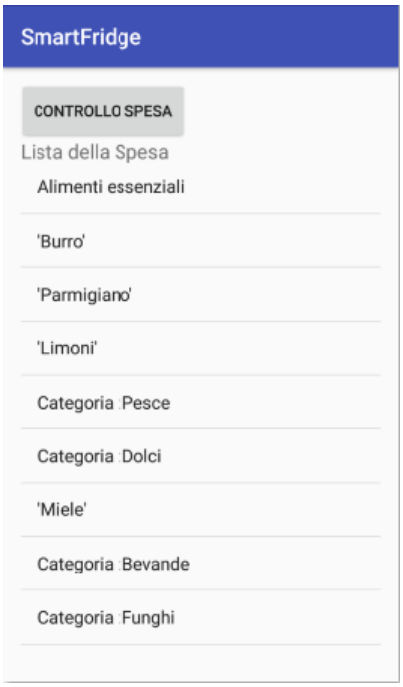

Figura 3.14: Activity per richiedere e mostrare la lista della spesa.

Registrazione client

Questa operazione è stata aggiunta per la necessità di fornire al FridgeServer i dati necessari per inviare ai vari utenti le liste delle ricette senza che siano loro a richiederle.
L'utente semplicemente esprime la sua intenzione di registrarsi al server il quale salva in un file, interrogabile da tuProlog, il suo indirizzo IP assieme ad un identificatore univoco per tale client, che viene poi mandato in risposta all'applicazione Android.
L'utente potrà decidere di cancellare la sua registrazioni in qualsiasi momento.

```
public void addClient(String idClient,String ipClient) {
    Theory nTheory;
    String clientToAdd = "client('"+idClient+"','"+ipClient+"').";
    FileWriter fw;
```

```
    try {
        engine.clearTheory();
        fw = new FileWriter(clientsPath, true);
        BufferedWriter bw = new BufferedWriter(fw);
        PrintWriter out = new PrintWriter(bw);
        out.println(clientToAdd); //Aggiunta del nuovo client
        out.close();
        nTheory = new Theory(new FileInputStream(frigoDBPath));
        engine.setTheory(nTheory);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
public void removeClient(String idClient,String ipClient) {
    Theory nTheory;
    String clientToRemove = "client('"+idClient+"','"+ipClient+"').";
    try {
        nTheory = new Theory(new FileInputStream(clientsPath));
        engine.setTheory(nTheory);
        SolveInfo info = engine.solve(clientToRemove);
        if(info.isSuccess()) {
            //Rimozione dell'entry del client specificato
            alterLine(clientToRemove,null,clientsPath,true);
        }
        nTheory = new Theory(new FileInputStream(frigoDBPath));
        engine.setTheory(nTheory);
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

L'activity android per effettuare questa operazione è accessibile tramite il menu presente all'interno dell'activity principale.

All'interno di questa activity sono presenti due campi di testo, uno modificabile e l'altro no, nel primo l'utente dovrà inserire l'indirizzo ip del server in modo che l'applicazione possa connettersi (nel caso della mancanza di un indirizzo IP statico), tale indirizzo ip sarà salvato una volta premuto il bottone "Set new server IP".
Il secondo campo di testo invece serve a contenere l'ID inviato dal server una volta che viene completata la procedura di registrazione, richiesta tramite il bottone "Register to server".
Il pulsante "Unregister" invece invia al server l'intento di de-registrarsi, e l'avvenimento sarà confermato dall'apparizione della scritta "Non Registrato" all'interno del campo di testo che conteneva l'ID.

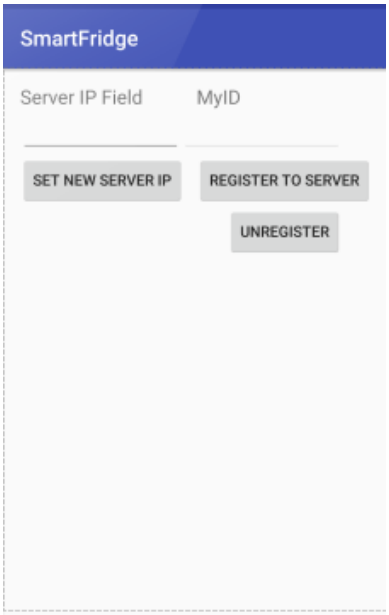

Figura 3.15: Activity per la registrazione del client.

Modifica dati personali

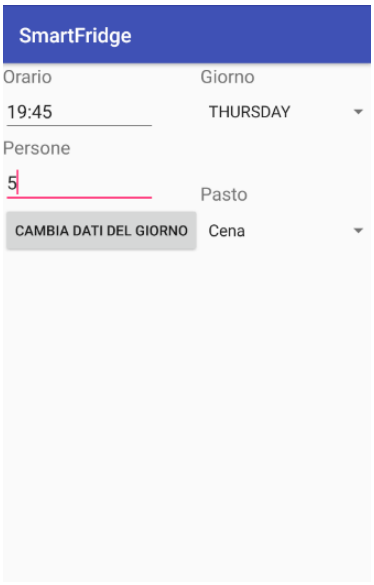

Figura 3.16: Activity per la modifica dei dati relativi ai pasti.

Per utilizzare questa funzionalità l'utente dovrà specificare, attraverso gli Spinner dell'interfaccia, il giorno e il pasto che vuole modificare, inserendo nelle due caselle di testo il nuovo orario e il nuovo numero di persone che saranno presenti.
Alla pressione del pulsante "Cambia dati del giorno" i dati verranno inviati al server che modificherà l'occorrenza specificata all'interno della Knowledge Base.
Nel caso l'utente voglia modificare solo uno dei due valori, basterà che lasci vuota la casella di testo e il meccanismo ignorerà tale parametro, lasciando inalterato il valore precedente.

```
public void changeDay(String newPersone,String newPasto,String
    newDay,String newOrario) {
  engine.clearTheory();
  Theory nTheory;
  try {
```

```
            nTheory = new Theory(new FileInputStream(daysPath));
            engine.setTheory(nTheory);
            SolveInfo info =
                engine.solve("meal('"+newPasto+"',"+newDay+"',Y,X).");
            String persone = info.getVarValue("X").toString();
            String oldOrario = info.getVarValue("Y").toString();
            oldOrario = oldOrario.substring(1, oldOrario.length()-1);
            persone = persone.substring(1, persone.length()-1);
            String oldLine =
                "meal('"+newPasto+"',"+newDay+"',"+oldOrario+"',"+persone+").";
            //Costruzione della nuova stringa
            String newLine = "meal('"+newPasto+"',"+newDay+"',";
            if(newOrario.equals("")) {
                newLine = newLine.concat("'"+oldOrario+"',");
            }else {
                newLine = newLine.concat("'"+newOrario+"',");
            }
            if(newPersone.equals("")) {
                newLine = newLine.concat("'"+persone+").");
            }else {
                newLine = newLine.concat("'"+newPersone+").");
            }
            alterLine(oldLine,newLine,daysPath,false); //Modifica dei dati del
                giorno

            nTheory = new Theory(new FileInputStream(frigoDBPath));
            engine.setTheory(nTheory);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Aggiunta delle nuove ricette

L'utente avrà la possibilità di aggiungere delle ricette personalizzate alla Knowledge Base.

Figura 3.17: Activity per l'aggiunta delle ricette.

Per fare ciò dovrà inserire all'interno dei campi di testo dell'interfaccia utente: il nome della ricetta che si vuole inserire, il pasto a cui sarà assegnata e una lista di ingredienti specificati dal loro tipo e categoria.
Per aggiungere un ingrediente alla lista sarà necessario specificarne il tipo all'interno della casella di testo, e poi sceglere una categoria tra quelle presenti all'interno dello spinner (questo poichè la Knowledge Base sfrutta tali categorie predefinite per la ricerca degli alimenti), confermando poi l'inserimento con il pulsante "Aggiungi ingrediente".
Premere il bottone "Aggiungi Ricetta" invierà al server tutti i dati inseriti, che poi verranno utilizzati per comporre una nuova ricetta, scritta come una nuova Regola tuProlog, impostata esattamente come quelle predefinite.

La nuova ricetta sarà disponibile nelle ricerche future, se i dati degli ingredienti sono in linea con quelli definiti dalla Knowledge Base.

```
public void addRecipe(String nome,String pasto,int
    nIngredients,ArrayList<String> ingredients) {
  String newRecipe = "recipe('"+nome+"','"+pasto+"',[";
  //Si inseriscono gli ingredienti nella lista
  for(int i=0;i<nIngredients;i++) {
    if(i!=nIngredients-1) {
      String idF = "ID"+i+",";
      newRecipe = newRecipe.concat(idF);
    }else {
      String idF = "ID"+i+"]) :-";
      newRecipe = newRecipe.concat(idF);
    }
  }
  //Si iterano gli ingredienti, inserendo all'interno della regola tutte
      le clausole necessarie
  for(int n=0;n<nIngredients;n++) {
    String ingrediente = ingredients.get(n);
    String[] ingr = ingrediente.split(",");
    String nomeIngr = ingr[0].trim();
    String catIngr = ingr[1].trim();
    if(n!=nIngredients-1) {
      newRecipe = newRecipe.concat("food(ID"+n+",
          '"+nomeIngr+"','"+catIngr+"',N"+n+"),");
    }else {
      newRecipe = newRecipe.concat("food(ID"+n+",
          '"+nomeIngr+"','"+catIngr+"',N"+n+").");
    }
  }
  engine.clearTheory();
  FileWriter fw;
```

```
        try {
            fw = new FileWriter(frigoDBPath, true);
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter out = new PrintWriter(bw);
            out.println(newRecipe);
            out.close();

            Theory nTheory = new Theory(new FileInputStream(frigoDBPath));
            engine.setTheory(nTheory);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Gestione delle connesioni con i client

Le connessioni con i singoli client sono gestite interamente dal processo principale dell'applicazione FridgeServer, il quale una volta ricevuta una richiesta di connessione delega l'intera operazione di servire il client ad un thread separato, così da poter tornare immediatamente ad accettare altre connessioni.
Il thread utilizzato per servire i client ottiene prima di tutto il flag, che viene sempre inviato come primo elemento di una lista, per sapere quale operazione deve effettuare.
Una volta capito quale operazione è richiesta, il thread procede cercando di ottenere il lock per le risorse necessarie, bloccandosi nel caso le risorse siano già in uso, e una volta ottenuto procede con l'utilizzo dei metodi necessari, rilasciando poi il lock alla fine del processo.

```
try {
        accepter = new ServerSocket(9999);
        while (accepting) {
```

```
            Socket socket = accepter.accept();
            new ClientResponseThread(
                socket, clients ,prologLock,clientLock,prologMethods).start();

        }
        accepter.close();
    }catch(Exception e) {
        e.printStackTrace();
    }
}
```

```
public class ClientResponseThread extends Thread {

    protected List<Client> clients;
    protected PrologLock clientLock,prologLock;
    protected Socket socket;
    protected ObjectInputStream in;
    protected ObjectOutputStream out;
    protected PrologEngine prologMethods;


    public ClientResponseThread(Socket socket,List<Client> clients,PrologLock
         prologLock,PrologLock clientLock,PrologEngine prologMethods) {
        this.clients = clients;
        this.socket = socket;
        this.prologLock = prologLock;
        this.clientLock = clientLock;
        this.prologMethods = prologMethods;
    }

    public void run() {
        try {
            boolean existing = false;
            in = new ObjectInputStream(socket.getInputStream());
```

```
List<String> received = (List<String>) in.readObject();
String request = received.get(0);

if (request.equals("UNREGISTER")) {

    clientLock.lock();
    String id = received.get(1);
    for (int i=0;i<clients.size();i++) {
        if ( clients.get(i).getID().equals(id)) {
        prologMethods.removeClient(
            clients.get(i).getID(), clients.get(i).getIP());
            clients.remove(i);
            break;
        }
    }
    clientLock.unlock();

}else if (request.equals("GET_RECIPES")) {
    prologLock.lock();
    ArrayList<String> recipeList = prologMethods.searchRecipes();
    out = new ObjectOutputStream(socket.getOutputStream());
    out.writeObject(recipeList);
    out.flush();
    prologLock.unlock();
}else if (request.equals("GET_SPESA")) {
    ArrayList<String> essentialsList =
        prologMethods.checkMissingEssentials();
    ArrayList<String> spesaList = null;
    boolean needToSpesa = prologMethods.checkSpesa();
    if (needToSpesa) {
        spesaList = prologMethods.getSpesaList();
    }
    essentialsList.addAll(spesaList);
```

```
        out = new ObjectOutputStream(socket.getOutputStream());
        out.writeObject( essentialsList );
        out.flush();
        out.close();
    }else if(request.equals("CAMBIO_GIORNI")) {
        prologLock.lock();
        String newPasto = received.get(1);
        String newDay = received.get(2);
        String newOrario = received.get(3);
        String newPersone = received.get(4);
        prologMethods.changeDay(newPasto,newDay,newOrario,newPersone);
        prologLock.unlock();
    }else if(request.equals("REGISTER")) {
        clientLock.lock();
        String ip = received.get(1);
        String id = "ID"+(clients.size());
        Client newClient = new Client(id,ip);
        out = new ObjectOutputStream(socket.getOutputStream());

        for(int i=0;i<clients.size();i++) {
            if( clients.get(i).getIP().equals(ip)) {
                existing = true;
            }
        }
        if(!existing) {
            clients.add(newClient);
            prologMethods.addClient(id,ip);
            System.out.println(""+newClient.getID());
            out.writeObject(newClient.getID());
        }else {
            for(int i=0;i<clients.size();i++) {
                if( clients.get(i).getIP().equals(newClient.getIP())) {
                    out.writeObject(clients.get(i).getID());
```

```
                    break;
                }
            }
        }
        clientLock.unlock();
        out.flush();
        out.close();
    }else if(request.equals("ADD_RECIPE")){

        String nomeRicetta = received.get(1);
        String nomePasto = received.get(2);
        ArrayList<String> ingredientList
            =(ArrayList<String>)in.readObject();
        int nIngredients = ingredientList.size();
        prologLock.lock();
        prologMethods.addRecipe(nomeRicetta, nomePasto, nIngredients,
            ingredientList);
        prologLock.unlock();
    }
    in.close();
    socket.close();
  }catch(Exception e) {
    e.printStackTrace();
  }
 }
}
```

Invio automatico a tutti i client

Al primo avvio il processo principale, prima di mettersi in ascolto per le connessioni dei client, il sistema programmerà un Task che si occuperà di inviare la lista delle ricette per il prossimo pasto programmato a tutti i client registrati, seguendo gli stessi meccanismi di ricerca descritti in precedenza.

Le applicazioni Android rimarranno sempre in attesa di queste connessioni tramite un servizio in background, che ad avvenuta connessione porterà immediatamente l'attenzione dell'utente sulla nuova lista di ricette ricevuta, forzando l'interfaccia principale in foreground.
Se l'applicazione è chiusa, oppure in background, il servizio rimarrà comunque in funzione, immagazzinando la lista delle ricette all'interno delle SharedPreferences che verranno recuperate al prossimo avvio dell'applicazione, o al suo ritorno al foreground.
Dopo aver inviato la lista, il Task otterrà dalla Knowledge Base l'orario del prossimo pasto e programmerà nuovamente un nuovo Task per essere eseguito all'orario rilevato, mantenendoì cos un ciclo continuo.

```
public FridgeServer(PrologLock prologLock,PrologLock clientLock) {

    prologMethods = new PrologEngine();
    acceptSensors = new AcceptSensorsThread(prologLock,prologMethods);
    acceptSensors.start();
    this.prologLock = prologLock;
    this.clientLock = clientLock;
    accepting = true;

    Calendar cal = Calendar.getInstance();
    ZonedDateTime time = ZonedDateTime.now();
    String day = time.getDayOfWeek().toString();
    cal.set(Calendar.HOUR_OF_DAY,1);
    cal.set(Calendar.MINUTE,0);
    Date firstDate = cal.getTime();
    String pasto = prologMethods.getPastoForInitialTask();
    clients = prologMethods.getClientFromProlog();
    Timer timer = new Timer();
    timer.schedule(new toAndroidTimedTask
        (prologLock,clientLock,clients,prologMethods,timer,pasto,day),firstDate);
```

```
public class toAndroidTimedTask extends TimerTask {

    protected PrologLock prologLock,clientLock;
    protected List<Client> clients;
    protected PrologEngine prologMethods;
    protected Timer timer;
    protected String pasto,day;

    public toAndroidTimedTask(PrologLock prologLock,PrologLock
        clientLock,List<Client> clients,PrologEngine prologMethods,Timer
        timer,String pasto,String day) {
        this.prologLock = prologLock;
        this.clientLock = clientLock;
        this.clients = clients;
        this.prologMethods = prologMethods;
        this.timer = timer;
        this.pasto = pasto;
        this.day = day;
    }

    public void run() {
        try {
            Socket clientSocket;
            ObjectOutputStream out;
            Date nextDate;
            prologLock.lock();
            ArrayList<String> recipeList = prologMethods.searchRecipes();
            clientLock.lock();
            for(int i= 0;i<clients.size() & clients.size()!=0;i++) {
                clientSocket = new Socket(clients.get(i).getIP(),9998);
                out = new ObjectOutputStream(clientSocket.getOutputStream());
                //Invio lista delle ricette ad ogni client
                out.writeObject(recipeList);
```

```
                out.flush();
                out.close();
                clientSocket.close();
            }
            clientLock.unlock();
            //Ri-programmazione del Task
            if(pasto.equals("Colazione")) {
                nextDate = prologMethods.getDateForNextTask("Pranzo",day);
                timer.schedule(new toAndroidTimedTask(prologLock,
                    clientLock,clients,prologMethods,timer,"Pranzo",day),nextDate);
            }else if(pasto.equals("Pranzo")) {
                nextDate = prologMethods.getDateForNextTask("Cena",day);
                timer.schedule(new toAndroidTimedTask(prologLock,
                    clientLock,clients,prologMethods,timer,"Cena",day),nextDate);
            }else if(pasto.equals("Cena")) {
                String newDay = prologMethods.getNextDay(day);
                nextDate =
                    prologMethods.getDateForNextTask("Colazione",newDay);
                timer.schedule(new toAndroidTimedTask(prologLock,
                    clientLock,clients,prologMethods,timer,"Colazione",newDay),nextDate);
            }
            prologLock.unlock();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Conclusioni

Il paradigma della programmazione logica è in grado di fornire all'Internet of Things la forma di intelligenza richiesta, sotto la forma di piccoli engine logici sparsi nell'ambiente e legati ai singoli oggetti che rappresentano.
Questo è soprattutto reso possibile grazie alle caratteristiche di portabilità e minimalità di tuProlog che permettono di realizzare questi engine su moltissimi dispositivi, senza incappare in problemi legati alla performance.
Il caso di studio visto all'interno di questa tesi, lo SmartFridge, potrà essere esteso in futuro realizzando una vera e propria cucina intelligente, creando altri engine logici per tenere traccia di tutti gli elementi presenti all'interno di una cucina, sfruttando le stesse architetture mostrate all'interno dell'elaborato.

Bibliografia

[1] Hakima Chaouchi. The Internet Of Things, Connecting Objects to the Web. ISTE, 2010.

[2] L. Atzoria, A. Ierab and G. Morabito, "The Internet of Things: A survey". Computer Networks. Vol. 54, issue 15. (2010).

[3] Arsénio, Artur Miguel & Serra, Hugo & Francisco, Rui & Nabais, Fernando & Andrade, João & Serrano, Eduardo. (2014). Internet of Intelligent Things: Bringing Artificial Intelligence into Things and Communication Networks. Studies in Computational Intelligence.

[4] International Telecommunication Unit. The Internet of Things, Strategy and Policy Unit. November 2005.

[5] R.A. Kowalski. Predicate logic as a programming language. In Proceedings IFIP'74. North-Holland, 1974.

[6] A. Colmerauer and Ph. Roussel. The birth of Prolog. In Thomas J. Bergin and Richard G. Gibson, editors, History of Programming Languages. ACM Press/Addison-Wesley, 1996.

[7] Apt, Krzysztof. (2001). The Logic Programming Paradigm and Prolog.

[8] tuProlog. Home page. http://tuprolog.unibo.it.

[9] Denti, Enrico & Omicini, Andrea & Calegari, Roberta. (2013). tuProlog: Making Prolog Ubiquitous.

[10] Denti, Enrico & Omicini, Andrea & Ricci, Alessandro. (2005). Multi-paradigm Java-Prolog integration in tuProlog. Science of Computer Programming. 57. 217-250.

[11] Calegari, Roberta & Denti, Enrico & Mariani, Stefano & Omicini, Andrea. (2017). Logic Programming as a Service (LPaaS): Intelligence for the IoT.