Sistema di regole per la configurazione hardware e la programmazione di applicazioni IoT

Tesi di Laurea Magistrale in Ingegneria Informatica

Antonio Montalto

Relatore: Prof. Daniele Peri
Correlatore: Ing. Gloria Martorella

UNIVERSITÀ DEGLI STUDI DI PALERMO
DIPARTIMENTO DELL'INNOVAZIONE INDUSTRIALE E DIGITALE
INGEGNERIA CHIMICA, GESTIONALE, INFORMATICA, MECCANICA

SCUOLA POLITECNICA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Sistema di regole per la configurazione hardware e la programmazione di applicazioni IoT

TESI DI LAUREA DI
Dott. Antonio Montalto

RELATORE
Ch.mo Prof. Daniele Peri

CONTRORELATORE

CORRELATORE
Dott. Ing. Gloria Martorella

ANNO ACCADEMICO 2016 – 2017

Sistema di regole per la configurazione hardware e la programmazione di applicazioni IoT

Tesi di Laurea di
Dott. Antonio Montalto

Relatore:
Ch.mo Prof. Daniele Peri

Controrelatore:

Correlatore:
Dott. Ing. Gloria Martorella

Sommario

I continui sviluppi nel campo dell'internet degli oggetti (IoT) insieme da quelli nel campo dell'intelligenza artificiale che hanno caratterizzato questi ultimi decenni, oltre all'ampia ed incessante diffusione di dispositivi wireless di nuova generazione, offre nuove possibilità nello sviluppo di sistemi distribuiti, dati dalla presenza di un gran numero di nodi interconnessi e coordinati tra loro da agenti intelligenti, capaci di raggiungere obbiettivi anche complessi. In questa tesi si descrive lo sviluppo di un'agente intelligente utile, non solo a trovare soluzioni a problemi dipendenti dai vari ambiti d'uso gestendo aspetti come locazioni, oggetti e applicazione, ma anche a gestire tutte le possibili configurazioni relative a rete, hardware e canale di trasmissione, tramite la creazione di una base di conoscenza utile alla descrizione precisa, completa e non ambigua delle componenti. In tal modo gli utenti meno esperti nel campo della programmazione possono concentrarsi solo sull'obiettivo dell'applicazione, mentre quelli esperti possono ottimizzare i tempi di sviluppo dedicati a questa tipologia di sistemi. Si è realizzato un layer intermedio in Prolog così da nascondere i dettagli a basso livello relativi all'architettura dei singoli nodi ed i protocolli di comunicazione, sulla quale poter generare configurazioni valide ed istruzioni FORTH funzionanti così da potersi riferire agli elementi del problema ad alto livello. Per questo è necessario che l'agente intelligente possa fare affidamento su un sistema di regole adatte allo scopo.

Indice

1 Introduzione

L'Internet of Things (IoT) è la visione futura di Internet che contempla l'inclusione di un numero enorme di sistemi con l'obiettivo di fornire facile accesso a un'enorme quantità di dati sul mondo fisico, utili alla realizzazione di una rappresentazione simbolica dello stato del mondo su cui un agente intelligente, tramite appositi algoritmi di deduzione, ragionamento e problem solving, può pianificare delle sequenze di azioni utili al raggiungimento di uno stato desiderato, il tutto a pieno supporto e potenziamento di servizi e degli utenti che ne fanno uso. Molto è stato il lavoro svolto negli ultimi anni in tale contesto; è possibile osservare lo sviluppo di framework ed estensioni Python dedicati [1, 2], piattaforme per il calcolo distribuito in dispositivi dalle risorse limitate [3], così come l'integrazione dell'intelligenza artificiale per lo sviluppo di architetture capaci di garantire supporto all'adattamento ed all'auto gestione [4].

La disponibilità di architetture sempre più compatte, efficienti ed a basso costo, spesso caratterizzate da un basso livello di consumi e di risorse computazionali a disposizione – come ad esempio le schede Arduino o Raspberry Pi solo per citarne alcune tra le più diffuse – così come i progressi impressionanti derivanti dalla continua ricerca sulle reti di sensori wireless nelle piattaforme e nei servizi software [5-7] di cui si è dimostrata l'utilità e la praticità del rilevamento denso [8-10] ha comportato il continuo miglioramento delle piattaforme di cloud computing aprendo la strada a una serie non indifferente di nuove opportunità per applicazioni e servizi di Internet of Things; la disponibilità di oggetti interconnessi su piccola scala ha dato origine a promettenti ricerche che considerano come attori principali i dispositivi con risorse limitate in applicazioni su larga scala, con capacità di svolgere compiti, anche complessi, in piena cooperazione [11].

L'intera attività progettuale, descritta in parte dal seguente elaborato, consiste nello sviluppo di un sistema distribuito general-purpose caratterizzato dalla presenza di un elevato numero di nodi dalle svariate caratteristiche e capacità, insieme ad una componente centralizzata ospitante un sistema di regole in grado di interagire con altri nodi periferici – che hanno come funzione quella di interfacciarsi con l'ambiente esterno attraverso sensori e attuatori di varia natura – coordinandone tutte le operazioni utili al perseguimento di obiettivi desiderati, anche complessi.

In qualità di sistema general-purpose si tratta di un'implementazione capace di trovare facile applicazione in tutti i principali domini applicativi ed ambienti operativi interessati dallo sviluppo della IoT, quali ad esempio la domotica, robotica, biomedicale, monitoraggio in ambito industriale, telemetria, efficienza energetica, assistenza remota, tutela ambientale, rilevazione eventi avversi, smart city, settore terziario e molti altri.

L'idea generale è stata quella di utilizzare una piattaforma integrata basata su Prolog per l'implementazione efficiente di sistemi di controllo per dispositivi embedded. Come illustrato similmente in altri lavori [12], l'intera struttura del sistema di controllo la si può considerare come composta da due livelli distinti: il primo è il livello di controllo logico, in cui le procedure di controllo sono state implementate in Prolog, mentre il secondo fornisce tutti i mezzi necessari per la comunicazione sicura tra i diversi nodi facenti parte del sistema distribuito.

Questa tesi descrive il concetto di una piattaforma integrata che consente l'implementazione di routine di controllo basate su Prolog sulla quale si è realizzata una infrastruttura dotata di elevata personalizzazione (statica e dinamica, a seconda delle varie esigenze applicative), essere possibilmente autogestita, governare l'interazione di componenti e applicazioni, incapsulare l'intelligenza in forme adatte al loro sfruttamento nelle varie applicazioni. Per questo, la connettività e l'interoperabilità sono solo la base di questo progetto, ma fondamentali [13].

La seguente attività progettuale gode della piena collaborazione del Dott. Leonardo Giuliana che tramite l'elaborato parallelo, dal titolo "Simbologia per la configurazione hardware e la programmazione di applicazioni IoT", ne completa la descrizione, illustrando il lavoro svolto relativo allo sviluppo di un formalismo semplice, condiviso ed efficiente utilizzato dal sistema di regole e dai vari nodi del sistema distribuito per la trasmissione di stringhe simboliche contenenti informazioni e conoscenza in termini di dati, regole, codice e istruzioni da eseguire.

Con la presente si coglie l'occasione anche per ringraziare il Prof. Daniele Peri, Prof. Salvatore Gaglio, Prof. Giuseppe Lo Re, Prof.ssa Alessandra De Paola e la Dott. Ing. Gloria Martorella del corso di Ingegneria Informatica dell'Università Degli Studi di Palermo per il pieno supporto fornito durante lo sviluppo della progettualità in esame.

2 Il Sistema

Il sistema nella sua completezza richiama all'attenzione sia sulla componente software che su quella hardware, sia quella ad alto che a basso livello. L'hardware considerato, consiste di un elevato numero di dispositivi caratterizzati da un ridotto livello di risorse a disposizione – sia in termini di memoria che di prestazioni generali – che cooperano tra loro, scambiano conoscenza ed informazioni in termini di dati, regole e codice, utili al raggiungimento di obbiettivi ed il compimento di compiti anche complessi. La componente software è idealmente scomponibile in due ulteriori livelli principali, un (i) primo livello di controllo logico, in cui le procedure di controllo sono implementate in Prolog sotto forma di clausole di Horn, (ii) ed un secondo livello che fornisce mezzi per la comunicazione efficiente con i dispositivi particolari di cui è composto il sistema.

Per il progetto in esame è stata originariamente pianificata l'implementazione di tre componenti principali tra cui, (i) un sistema di regole, presente all'interno dell'unità centrale in grado di legare l'hardware con ambiente tramite la generazione di un codice simbolico - stringhe di configurazione ed operazioni, direttamente eseguibili dalle componenti secondarie – realizzato secondo una sintassi definita a partire dalle informazioni contenute all'interno di una base di conoscenza, (ii) un codice simbolico utile alla descrizione efficiente di informazioni, configurazioni ed operazioni, ovvero una sequenza di simboli, secondo un determinato formalismo, da fare eseguire ai vari componenti del sistema distribuito (iii) ed infine una GUI (graphical user interface) user-friendly realizzata per mezzo di una Web Application, ospitata anch'essa all'interno della componente centrale del sistema, con il quale l'utente finale ha la possibilità di monitorare e controllare tutti gli aspetti gestiti dal sistema ed i vari parametri relativi ai nodi ed all'ambiente su cui opera.

Nell'approccio presentato in questa tesi, l'intera logica del sistema di controllo è stata scritta in Prolog. Tale linguaggio si è rivelata una buona scelta grazie alle sue caratteristiche avanzate come l'alta espressività e la rappresentazione della conoscenza basata sulla logica concettuale con cui la conoscenza dell'esperto può essere trasferita all'interno del sistema senza richiedere particolari conoscenze informatiche da parte dell'utente. Tuttavia, vi sono alcuni seri ostacoli nell'utilizzo di Prolog per l'implementazione pratica dei sistemi di controllo della vita reale.

Il problema principale è che Prolog è a tutti gli effetti un linguaggio di alto livello e le implementazioni generiche non forniscono mezzi affidabili per la comunicazione e il controllo dell'hardware, specie quello trattato in questa attività. Ed è per questo il motivo che all'interno del modello, oltre alla rappresentazione del problema, vi è una parte che ha il compito di integrare tra loro, il livello di controllo del problema con il livello inferiore dedicata alla rappresentazione e la gestione della parte hardware del sistema.

Il risultato finale, per quanto riguarda l'intero sistema di regole, è quello di un sistema caratterizzato da elevata personalizzazione, capacità di autogestione e interazione con le componenti e che, attraverso l'analisi del modello del problema e dell'ambientale – nonché il suo stato di partenza – è in grado di dedurre il percorso più efficiente da compiere utile all'ottenimento dell'obiettivo desiderato – ovvero il suo stato finale – il tutto attraverso la generazione di stringhe simboliche idonee da trasmettere ai vari nodi del sistema.

Le stringhe simboliche generate sono istruzioni FORTH; si tratta di un linguaggio di programmazione procedurale, general-purpose, dalla grammatica molto semplice ed in grado di dare ampia libertà allo sviluppatore della quale prima apparizione si ha nel 1972, presentato come strumento di sviluppo che mira a sfruttare in modo efficiente l'hardware. Un programma FORTH si presenta come una sequenza di simboli ad alto livello, dette *words*, definite nel dizionario, e costanti letterali, di cui meccanismo di valutazione segue la notazione polacca inversa (RPN). Le voci del dizionario contengono il nome della *word* più altre utili informazioni, incluso il codice da eseguire quando la *word* viene interpretata, inoltre, FORTH è dotato di meccanismi che permettono di ampliare il numero delle voci all'interno del dizionario; possono essere definite nuove *word* come sequenze di *word* già definite, o anche *word* built-in.

All'interno dei dispositivi interessati, oltre all'interprete FORTH, è già presente parte del codice simbolico necessario così da garantire una base di compatibilità tra i nodi del sistema, mentre una seconda parte sarà generata dinamicamente dal sistema di regole sulla base della combinazione tra configurazione del sistema ed i risultati desiderati dall'utente. Il codice FORTH ottenuto a partire da tale processo di pianificazione, svolto sulla base del modello dell'ambiente, viene quindi inviato ai vari dispositivi per l'esecuzione attraverso un protocollo di comunicazione sicuro.

Un possibile esempio di attività svolta dal seguente sistema, considerandone l'applicazione nel contesto della domotica, può essere quello di variare i livelli di luminosità o di temperatura di una determinata stanza: il compito del sistema sarà quello di rilevazione i livelli di luminosità e di temperatura in quel momento presenti nell'ambiente interpellando unicamente i nodi dotati di appositi sensori adatti allo scopo, in seguitò confronterà i valori rilevati dai nodi con quelli indicati dall'utente e, nel caso risultassero non idonei, il sistema provvederà alla generazione ed all'invio di istruzioni precise a tutti o anche solo ad alcuni dei nodi in grado di agire sugli oggetti capaci di adeguare lo specifico parametro fisico nella direzione desiderata (Es. lampade e serrande possono agire sul parametro di luminosità, così come condizionatori e stufe possono agire sul parametro temperatura e così via) presenti nella sola locazione indicata, il tutto sempre con lo scopo di raggiungere l'obiettivo desiderato dall'utente.

Il sistema di regole, si occupa di eseguire tutti i processi di pianificazione necessari, basando le sue ricerche su un database deduttivo, di cui parte del suo contenuto è variabile dipende dal contesto d'uso e dall'obiettivo della ricerca, mentre una seconda parte statica rappresenta lo strato che determina la compatibilità con l'hardware embedded. Con l'ausilio del database deduttivo è in grado di gestire le informazioni relative al problema da risolvere, l'ambiente ed il contesto in cui si presenta, ed i passi da svolgere oltre che provvedere alla configurazione hardware di tutti i nodi i sensori e attuatori di cui è dotata (specie dei collegamenti) più altre informazioni generali ad esse relative come le loro locazioni all'interno dell'ambiente, oltre che – ovviamente – le eventuali preferenze specificate dall'utente.

3 Sistema di regole e la base di conoscenza

Come già anticipato, l'agente intelligente artificiale può fare affidamento su un database deduttivo contenente un insieme di fatti e regole, sotto forma di clausole di Horn, idealmente scomponibili in due ulteriori categorie: (i) la prima è relativa agli elementi dell'ambiente, variabile a seconda del contesto e del problema – può essere uno tra quelli indicati nell'introduzione o altro – da cui il processo di pianificazione ricava le operazioni necessarie per raggiungere l'obiettivo, (ii) e la seconda – indipendente dall'ambito d'utilizzo, ma più orientata verso l'hardware – garantisce elevati livelli di configurabilità ed interconnessione tra i vari nodi a disposizione del sistema, attraverso la capacita di generare delle stringhe simboliche da inviare ad ognuno di essi tramite i diversi protocolli di interconnessione definiti. È inoltre compito del sistema quello di garantire all'utilizzatore finale la segretezza delle informazioni scambiate, l'autenticità delle informazioni ed il controllo degli accessi, il tutto attraverso l'uso di un canale di comunicazione sicuro per tutte le trasmissioni necessarie.

A scopo sperimentale è stato considerato come contesto d'uso quello della domotica, quindi sono stato portati avanti diversi test per la verifica del corretto funzionamento in un ambiente idoneo alla rappresentazione del suddetto contesto così come dei tipici problemi che lo possono caratterizzare.

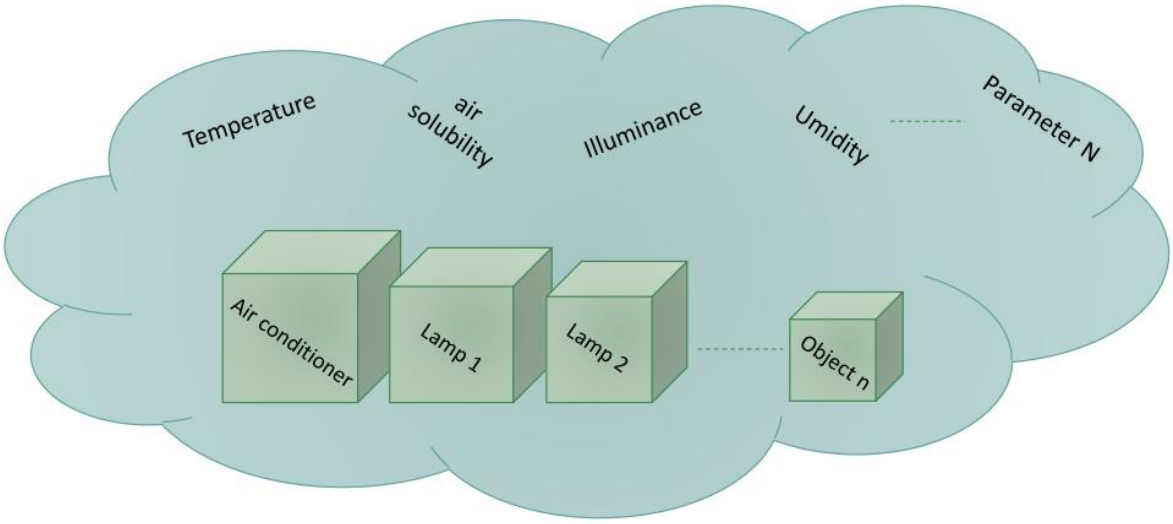


Fig. 1 Elementi della base di conoscenza dell'ambiente

L'analisi dell'ambiente la si può considerare come facente parte del livello applicativo (*Application Level*) e prende in esame aspetti come le locazioni, gli oggetti, i parametri fisici ambientali ed eventuali azioni dedicate. Se una parte della base di conoscenza è strettamente legata alla descrizione del problema specifico (descrizione di cui l'esperto si è fatto carico di rappresentare correttamente all'interno della base di conoscenza), una seconda parte della base di conoscenza è composta dai fatti e dalle regole relative all'ambiente che possono risultare indipendenti da uno specifico contesto d'uso, o comunque risultare d'uso comune a più contesti.

Nella seguente attività di progetto si sono identificate le seguenti componenti per cui è risultata necessaria un'attenta rappresentazione all'interno della base di conoscenza:

1) Le locazioni: luoghi generici o specifiche posizioni nello spazio.
2) Gli oggetti: tutti i dispositivi che si possono trovare dislocati nelle varie locazioni e che possono essere gestiti dal sistema.
3) I parametri e gli stati: l'insieme di tutti gli aspetti e parametri fisici che descrivono lo stato dell'ambiente, come il livello di luminosità, lo stato degli oggetti dell'ambiente ed altro.
4) L'hardware: si vogliono descrivere tutte le caratteristiche dell'hardware utilizzato come possibili candidati nel ruolo di nodi di sistema.

Da questo momento in avanti si procederà ad una descrizione dettagliata di tutti gli aspetti e le difficoltà di cui si è dovuto tenere conto per la corretta creazione di una base di conoscenza capace di gestire in maniera idonea tutti gli aspetti relative alle componenti sopra citate.

Mentre la descrizione dell'hardware si può considerare "costante", poiché indipendente dal contesto e dal problema da affrontare, le informazioni relative alle locazioni, gli oggetti ed i parametri possono variarne nei contenuti se non addirittura essere completamente assenti poiché ininfluenti ai fini del contesto d'uso e dei problemi da risolvere tramite l'azione di questo sistema. Le variazioni sulla base di conoscenza relativa all'hardware possono trovare comunque giustificazione ai fini di aggiungere o rimuovere supporto (intesa come capacità di gestione) a nuovi e vecchi dispositivi embedded, così come correggere o integrare le informazioni relative alle funzionalità di quelle già presenti.

3.1 Locazioni, oggetti e stato

Come già anticipato una locazione indica una posizione dell'ambiente considerato. Essa può contenere al suo interno altre locazioni più specifiche fino ad arrivare a quelle contenenti strettamente i singoli oggetti che si vuole gestire; tale impostazione suggerisce per la sua descrizione l'uso di una struttura ad albero in cui come radice si ha la locazione più generale – può essere "Casa", nel contesto d'uso della domotica preso in considerazione – mentre nei nodi interni si identificano locazioni sempre più specifiche raggiungibili dal rispettivo padre, come le stanze e/o specifiche posizioni all'interno di esse, infine le foglie identificano gli oggetti che si possono trovare nelle varie posizioni; in questo modo si può tracciare la posizione dei vari oggetti tracciando il percorso che porta dalla foglia (l'oggetto) fino alla radice, o viceversa, ricavare la lista di tutti gli oggetti presenti all'interno di una determinata location partendo da un particolare nodo intermedio a qualsiasi livello (una locazione) e seguendo tutte le diramazioni possibili fino al raggiungimento delle foglie. Prolog viene perfettamente incontro alla risoluzione del suddetto problema nella maniera più semplice ed efficiente possibile.

La suddetta struttura ad albero – realizzata tramite clausole di Horn in Prolog – può essere interrogata tramite opportuni processi di risoluzione portando alla generazione della sintassi generica, cioè i comandi che uno o più nodi può eseguire sui dispositivi o sull'ambiente, riuscendo a garantire, nei vari riferimenti ai nodi del sistema distribuito, univocità e generalità allo stesso tempo.

All'utente viene infatti data la possibilità di rivolgersi al sistema attraverso le seguenti possibilità:

- È possibile riferirsi allo specifico dispositivo presente in una data locazione spaziale, anche la più generale (in tal è come non specificare la locazione in quanto la locazione più generale è sottointesa).
- Può riferirsi ad una specifica classe di dispositivo presente in una data locazione spaziale, anche la più generale (a riguardo valgono le stesse considerazioni presenti sopra), causando l'intervento di tutti gli oggetti di quella tipologia in grado di eseguire l'azione raggiungibili a partire dalla locazione specificata.
- Può anche riferirsi a tutti i dispositivi presenti all'interno di una locazione più generale (che ne racchiude altre), indipendentemente dalla tipologia, ma specificandone solo l'azione da eseguire, il tutto senza la necessità di doverli

indicare singolarmente all'agente intelligente; all'atto pratico possono essere interpellati tutti gli oggetti, anche se entrano in azione solo gli oggetti capaci di eseguire il comando, o meglio, quegli oggetti per cui l'azione risulta semanticamente equivalente un'azione valida: il comando di "*ON*" (inteso come accensione) lanciato nella locazione più generale (Es, "*casa*") causerebbe l'accensione di tutti i dispositivi al quale è stato legato quel comando, quindi oggetti come lampade e televisori si accenderebbero anche se tramite procedure FORTH molto diverse, la lampada potrebbe essere accesa da un relay, mentre il televisore dall'invio di un particolare segnale dal trasmettitore IrDA.

- Può specificare una lista di locazioni; in tal caso il sistema, nel processo di risoluzione sono considerate annidate l'una a l'altra (a qualsiasi livello e non per forza immediate sotto-locazioni); se venisse inviata un azione dove viene richiesto di accendere le lampade poste sui tavoli della casa (la lista sarebbe ["*casa*", "*tavolo*"]) andrebbero ignorate del tutto le lampade che nella gerarchia delle locazioni non trovano sopra a loro stesse le locazioni "*casa*" e "*tavolo*" (in questo ordine), di fatto ignorando le lampade poste in locazioni fuori dalla casa e non sopra un tavolo.

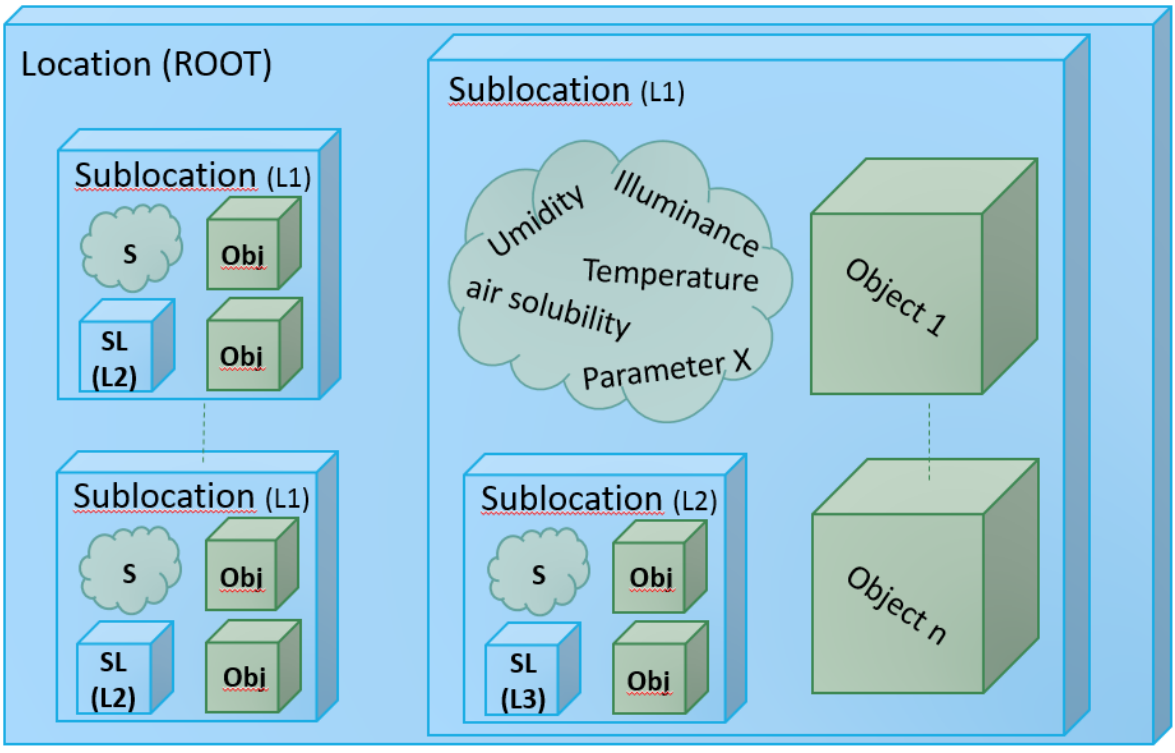


Fig. 2 Struttura ambiente, parametri, oggetti

È possibile, all'interno della struttura ad albero, che una locazione risulti come foglia, in quanto sotto di essa non è stata descritta alcuna sotto-locazione più specifica oppure inserito alcun oggetto (appelleremo tale configurazione come "non completa"); tuttavia ciò non costituisce un alcun problema per l'agente intelligente in quanto è stato messo nelle condizioni di saper distinguere tra le varie foglie se si tratta di locazioni, oggetti o nodi del sistema. Tale distinzione risulta essere fondamentale in quanto solo i vari oggetti e le periferiche presenti nel sistema sono interessati dalle eventuali azioni che l'agente intelligente ordina di eseguire, in quanto gli unici con le potenzialità di poterle compiere.

Tra i comandi inviabili dall'utente, a parità di contenuto semantico, è possibile distinguere diverse configurazioni; nello specifico da parte dell'utente è possibile definire almeno una o più locazioni (la logica artificiale le interpreta secondo l'ordine indicato come annidate tra di loro), a scelta, il parametro ambientale che si vuole modificare (lo stato) o gli oggetti (può essere la categoria di oggetti o un oggetto specifico) o ancora la specifica periferica (il dispositivo hardware, parte del sistema distribuito che controlla l'oggetto). Parte non opzionale del comando è l'azione che si desidera compiere tramite le componenti ricavate.

Tuttavia l'organizzazione della parte di base di conoscenza relativa all'ambiente descritta poc'anzi, non è da considerarsi immune ad ambiguità; è infatti compito dell'utente crea univocità nella generazione di percorsi oppure usare le caratteristiche univoche dell'oggetto o del suo percorso nei riferimenti generali agli oggetti così da evitare l'inclusione di oggetti e periferiche che non si desidera prendere in considerazione per quella particolare azione.

Questo problema tuttavia non è circoscritto al solo sistema di regole, ma anche alle persone nella vita reale; queste hanno imparato con il tempo e l'esperienza fatta a seguito di eventuali errori ad escludere le opzioni probabilmente errate, permettendo all'interlocutore di essere meno preciso nelle descrizioni; volendo fare un esempio pratico, a molti spesso capita di avere alla domanda – "dove si trovano le chiavi?" – una risposta come – "si trovano sul tavolo" – tuttavia, come spesso accade, non si ha un solo tavolo su cui cercare, nonostante questo, grazie all'esperienza, si è in grado di dare una sorta di priorità ai luoghi in cui si può cercare, con il risultato di riuscire a trovarle – almeno, nella maggior parte dei casi – al primo tentativo.

L’agente intelligente, però, a seconda delle indicazioni date, può fermarsi alla sola prima opzione “valida” per la semantica descritta dall’utente (non si intende per forza quella desiderata dall’utente), così come procedere a raccogliere tutte le opzioni valide (includendo possibilmente altre opzioni indesiderate); è quindi conveniente da parte dell’utente che la struttura descritta, così come i suoi riferimenti, contengano elementi che permettano all’agente intelligente di non cadere in indesiderate ambiguità descrittive. In questi casi, una soluzione parziale è quella di chiedere conferma all’utente di quali soluzioni escludere tra tutte quelle valide risultanti dalla definizione – eventualmente memorizzando i risultati per le richieste future – ma questo risulterebbe poco praticabile nel caso di azioni pianificate e poco efficiente nel caso di lancio di comandi diretti da parte dell’utente poiché questo, prima di vedere eseguita l’azione desiderata, si potrebbe ritrovare a dover dare conferma a parecchi elementi risultanti dal processo di estrapolazione.

Come illustrato nella figura sottostante, la base di conoscenza dell’ambiente viene continuamente tenuta aggiornata ed arricchita da ulteriori informazioni ottenute tramite interrogazioni dirette ai nodi del sistema dotati di sensori presenti nell’ambiente, grazie ad un meccanismo di feedback; alcune di queste informazioni sono ottenute “automaticamente” anche durante l’esecuzione di procedure che hanno un obiettivo diverso dal solo rilevamento dello stato ambientale.

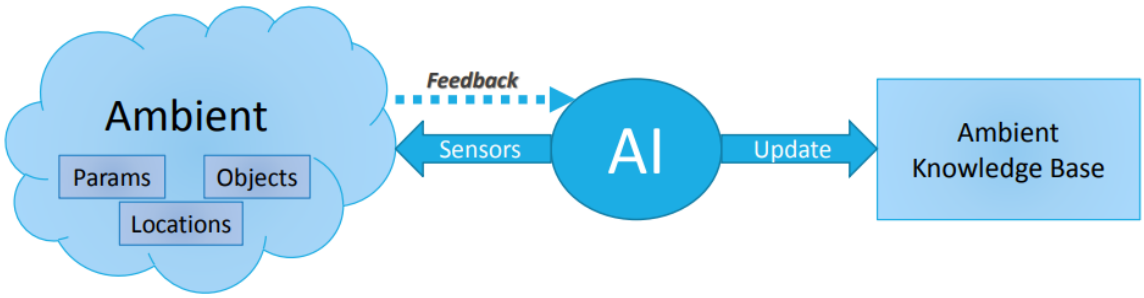


Fig. 3 Aggiornamento della base di conoscenza

3.2 Hardware

La seconda parte della base di conoscenza (*Board Level*) fa riferimento all'hardware – sia periferiche che MCU (MicroController Unit) – e si occupa di gestire gli aspetti di configurazione e generazione del codice da fare eseguire ai nodi del sistema ricavato tramite un processo di pianificazione che coinvolge anche la base di conoscenza relativa all'ambiente e gli obiettivi da perseguire, definiti implicitamente dal sistema sulla base di informazioni già presenti o sotto esplicita indicazione da parte dell'utente.

Questa parte del progetto rappresenta uno degli elementi più importanti dell'intero progetto e tratta argomenti già sotto esame in altri studi [14] dove si cerca di proporre delle metodologie di convalida basate su regole per la gestione di problemi di configurazione relativi all'hardware in modo efficiente.

Al di là del contesto d'uso della domotica presa in considerazione in questa attività, una base di conoscenza ed un sistema di regole in grado di gestire l'hardware e la sua configurazione contribuirebbe a ridurre le inefficienze che si hanno sviluppando su queste piattaforme; studi recenti dimostrano che il 50-70% del costo di sviluppo del software viene speso per la verifica, i test ed il debug [15], senza contare che con il passare del tempo aumentano le aspettative in termini di funzionalità che il software deve essere in grado di eseguire (secondo alcuni studi raddoppiano ogni dieci mesi [16]).

Per di più a contribuire a questo divario di produttività si aggiunge una mancanza di comprensione da parte degli sviluppatori di software su come l'hardware debba essere usato [17], infatti l'uso non corretto dell'hardware può comportare la presenza di bug funzionali (come il non funzionamento della trasmissione dati di una interfaccia di input/output), oppure di bug non funzionali come carenza di performance o un trasferimento dei dati sul bus inefficiente: questa tipologia di problematica risulta più difficile da risolvere e spesso non si riesce nemmeno a rilevarla in quanto spesso non si è nemmeno a conoscenza di quali dovrebbero essere i livelli di performance che si dovrebbero riuscire ad ottenere dal dispositivo hardware su cui si sta sviluppando.

Ciò non è necessariamente colpa degli sviluppatori: l'integrazione di un numero sempre maggiore di componenti in un singolo chip ha portato a regole complesse che regolano la configurazione e l'uso di questi componenti (solitamente configurati durante il runtime). L'aumento della complessità della configurazione è

illustrata al meglio dalla lunghezza dei manuali utente di riferimento dei SoC (system on a chip) di oggi, in grado di descrivere il corretto utilizzo dell'hardware: questi possono arrivare a diverse migliaia di pagine. Altro aspetto da considerare in relazione a questi manuali è che al loro interno è presente una descrizione scritto da altre persone nel linguaggio naturale, il che può dare origine a incomprensioni ed errori nel trasferire le regole d'uso del SoC hardware dal progettista allo sviluppatore software del chip.

Nella fase iniziale dello sviluppo SoC, i progettisti sono responsabili della definizione delle caratteristiche, delle metriche e delle prestazioni che deve avere. Inoltre, sono definite le regole del software di base che dovrebbero essere conformi al fine di ottenere il comportamento previsto. Invece di documentare queste regole in un modo convenzionale, ad es. manuale dell'utente, le regole possono essere formalizzate in un formato standardizzato. Questo sistema mette a disposizione un formato di regola standardizzato come input per una base di conoscenza, che funge da ponte tra la fase di progettazione del SoC e la fase di sviluppo del software necessario.

Tale metodologia per la generazione e convalida automatica della configurazione hardware – le cui regole sono riassunte direttamente dai progettisti di SoC e memorizzate nella base di conoscenza – oltre ad agevolare gli sviluppatori durante la convalida delle configurazioni hardware, aumentandone la produttività, gestisce delle regole che hanno il potenziale per essere convertite nel manuale utente definitivo secondo un formato matematico standardizzato facile da comprendere e privo di ogni ambiguità.

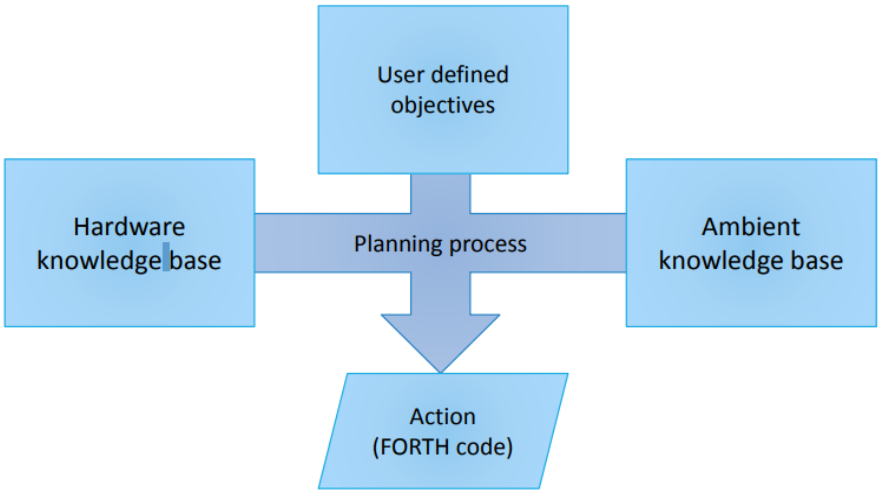


Fig. 4 Processo di pianificazione

La generazione della configurazione del sistema coinvolge aspetti come il tipo di connessione usata tra il nodo con l'unità centrale, così come il collegamento tra i sensori con la specifica MCU di cui è composto il nodo; quello di collegare correttamente un dato set di sensori con un determinato set di pin disponibili nella specifica MCU considerando tutti i possibili usi che essi possono coprire secondo quanto descritto nelle specifiche hardware, così come descritto in altri lavori dove si sono affrontate problematiche simili [18], è un contraint-satisfaction problem (CPS), ovvero un problema di soddisfazione dei vincoli. Si giunge facilmente alla conclusione che descrivere il CPS sotto forma di clausole di Horn in Prolog mette a disposizione una via intuitiva ed accessibile per gestire tutte le possibili configurazioni possibili che per configurazioni complesse possono raggiungere le diverse centinaia di migliaia se non anche milioni [18].

Per la risoluzione di tale problematica è necessario procedere alla descrizione delle specifiche di tutte le MCU che si vogliono rendere compatibili con il sistema – in termini di gestione e configurazione – e di tutti i loro pin compresi i molteplici usi che possono coprire, in modo analogo si esegue lo stesso censimento per tutti i sensori e attuatori che si desidera far trattare dal sistema, quindi si cerca di ricavare la configurazione migliore con cui collegare i diversi sensori – verificando prima la disponibilità, risolvendo eventuali conflitti (se possibile) e gestendo altri aspetti come i costi più bassi in termini di possibilità di copertura di un determinato compito –, ed infine si genera il codice richiesto per il corretto interfacciamento tra sensori con la MCU così come quello necessario per l'interfacciamento dell'intero nodo con l'unità centrale del sistema.

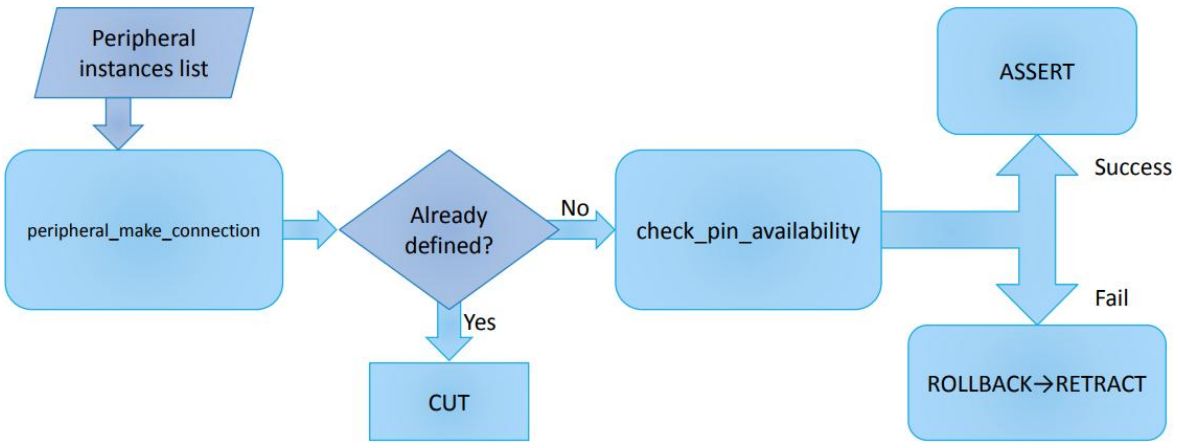


Fig. 5 Generazione di configurazioni di collegamento tra MCU e sensori/attuatori

Come illustrato nella figura 4, il sistema lanciando la regola Prolog `mcu_peripheral_make_connection/6`, controlla per prima cosa se esiste già una istanza di nodo con la definizione indicata dall'utente, e se non è stata già definita, provvede al controllo della disponibilità dei pin sulla MCU capaci di coprire il ruolo necessario al sensore o attuatore per il collegamento e in caso di successo, asserisce le nuove regole nella base di conoscenza relativa ai dispositivi ed i vari nodi presenti. È possibile anche specificare più sensori/attuatori contemporaneamente, così da trovare una configurazione valida per tutti gli elementi desiderati per lo specifico nodo; nella pratica l'agente intelligente esegue il controllo ed in caso asserisce la configurazione valida per il primo sensore/attuatore nella lista, quindi procede con il secondo, in questo modo si fa in modo che la configurazione di tutti gli elementi della lista tengano in considerazione i pin già occupati dagli elementi che l'anno preceduta nella configurazione.

La regola Prolog `mcu_peripheral_make_connection/6`, dedicata alla generazione della configurazione, richiede come parametro una lista composta dai seguenti elementi di connessione, di cui è possibile specificarne tutti o anche solo alcuni: istanza della MCU, pin della MCU (numero), funzionalità del pin della MCU, istanza del dispositivo (sensore/attuatore), pin del dispositivo (numero), funzionalità del pin del dispositivo. Nel caso di specifica manuale di tutti i parametri necessari, il sistema provvede a verificare la fattibilità di quella specifica connessione tra la data MCU ed il sensore/attuatore, utilizzando solo i pin indicati così come la funzionalità specificata dall'utente, il tutto senza prendere in considerazione alcuna alternativa. Nel caso di specifica parziale dei parametri di configurazione della connessione, il sistema provvede ad assegnare i parametri assenti automaticamente sulla base della disponibilità ed il supporto (in termini di funzionalità); nel caso non venisse specificato il pin tra i parametri di connessione, il sistema assegna il primo disponibile della lista capace di ricoprire la funzionalità (quest'ultima ricavata direttamente dalle informazioni già presenti nella base di conoscenza in quanto sono state inserite manualmente da parte del produttore o dall'utente ricavandole dalla documentazione disponibile o dai datasheet dei dispositivi).

Particolarmente importante è la regola `mcu_init/6`, tramite il quale l'utente ha la possibilità di inviare una lista composta da elementi della tipologia sopra menzionata (quindi una lista di liste di configurazione). Il sistema si occupa di verificare la fattibilità ed, in caso di esito positivo, di asserire le relative informazioni nella base di conoscenza, una per volta, tuttavia è bene specificare che il sistema tratta esclusivamente configurazioni complete.

Nel caso in cui l'agente intelligente non riesca a generare una configurazione valida per tutti gli elementi della lista – possibilmente dovuta alla mancanza di pin disponibili, capaci di ricoprire il ruolo richiesto dall'elemento – questo ritratta tutte le regole generate per il collegamento degli elementi che hanno trovato spazio nella MCU tramite apposite procedure di rollback definite. È sempre possibile per l'utente, "arricchire" ulteriormente la configurazione di nodi già definiti (sempre nel rispetto dei criteri di disponibilità definiti poc'anzi), donandone ulteriori capacità e possibilità nel sistema complessivo.

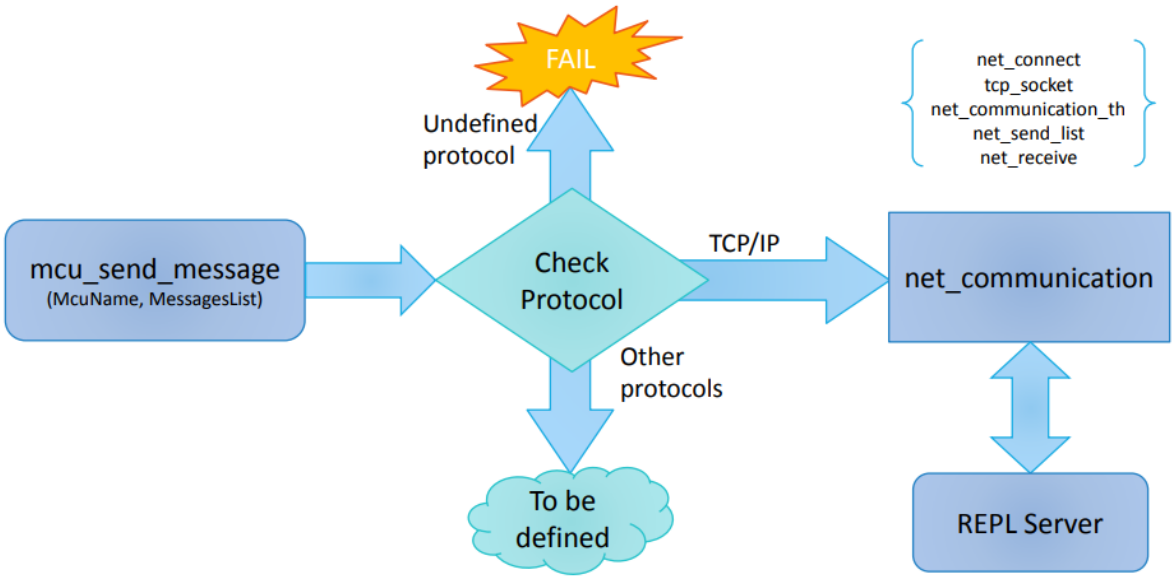


Fig. 6 Generazione della configurazione

I comandi ricavati a seguito del processo di pianificazione, vengono trasmessi ai dispositivi interessati tramite protocolli sicuri definiti nel sistema. Come illustrato nella figura 5, il sistema a seguito dell'esecuzione della regola Prolog `mcu_send_message`, che richiede come parametri il nodo da interpellare ed una lista di comandi da inviare alla MCU, effettua un controllo sul protocollo di comunicazione da utilizzare per l'invio dei comandi al nodo specifico e se riesce a trovarne uno valido procede con l'invio dei comandi alla MCU tramite la regola `net_communication`, che, nello specifico di quanto realizzato in questo progetto la rende eseguibile da parte di un server REPL (Read Eval Print Loop) già ospitato all'interno dei nodi.

A seguito del comando per l'invio delle istruzioni, `mcu_send_message/3` (sono disponibili diverse varianti della stessa regola distinte per arità: maggiori informazioni sono disponibili nel capitolo 8, paragrafo 1.3, del seguente elaborato, "l'implementazione") l'agente intelligente si aspetta anche un valore di ritorno dal nodo, più nello specifico una lista di elementi di dimensione pari alla lista dei comandi inviati; ad ogni posizione della lista delle risposte corrisponde, se previsto, un risultato nel vettore di risposta immagazzinato nella posizione equivalente a quella occupata dall'istruzione che la ha generata, in caso contrario, alla specifica posizione è presente l'elemento vuoto.

4 L’unità di controllo

Il nodo principale del sistema – detto anche unità centrale – si interfaccerà verso l'utente tramite un simbolismo semplice ispirato al linguaggio naturale e verso l'ambiente tramite i sensori e attuatori ad essa collegati direttamente – tramite le sue stesse GPIO se disponibili – o indirettamente tramite interfacciamento verso dei nodi esterni per mezzo di protocolli di varia natura quali ad esempio porte seriali, CAN bus, pin Ethernet, pin PWM, o ancora protocolli ad alto livello come il Wi-Fi o altro; l’agente intelligente trasmette istruzioni a tutti i nodi – eventualmente anche all’unità centrale stessa – in input e trasmetterà gli eventuali output prodotti (se richiesto) nell’interfaccia utente.

Per lo svolgimento dell’attività di tesi è stato preso in considerazione, nel ruolo di unità centrale, una piattaforma Raspberry-Pi, nello specifico il Pi uno modello B del 2012 munito di CPU ARM1176JZF-S single-core a 700 MHz e 512MB di RAM condivisa con la GPU Broadcom VideoCore IV e sistema operativo Raspbian 8.3.0 configurato ad-hoc per gli scopi progettuali.

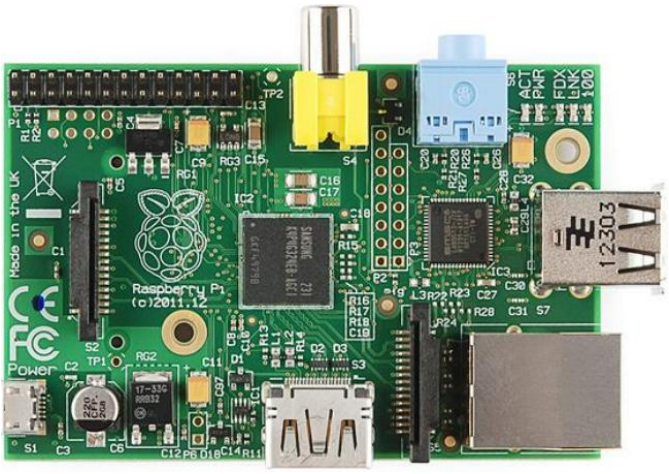


Fig. 7 Rasberry-Pi model B (2012)

Su di essa è stato fatto girare un server Apache 2.4 allo scopo di ospitare la GUI realizzata tramite i linguaggi di programmazione PHP (sono supportate le versioni 5.6 fino alla 7.2) e progettazione per il web (HTML, CSS, JavaScript) accompagnato da un database di appoggio (RDBMS Oracle MySQL) – utile a memorizzare eventuali regole e specifiche custom che dipendono da quanto appreso o specificato dall'utente – che si interfaccia con un interprete dedicato all'esecuzione del sistema di regole sviluppato in Prolog, oltre ad un interprete FORTH (nello specifico è stato scelto gForth per la l'unità centrale) per l'esecuzione di codice a basso livello per controllare i sensori e gli attuatori direttamente collegati ad essa tramite le porte GPIO.

La logica di controllo principale, contenuta in un programma dichiarativo Prolog, è eseguito da un compilatore Prolog (tutti gli "interpreti" Prolog avanzati precompilano il codice) e nello specifico ne è stata scelta una sua implementazione open source, SWI-Prolog; molte sono state le opzioni tra cui scegliere, come GNU Prolog, ognuna con le sue caratteristiche (es. CLP, Constraint Logic Programming, nel caso di GNU Prolog), tuttavia la scelta di usare come riferimento le API SWI-Prolog è giustificata dal fatto che è sembrato essere il più flessibile e ben consolidato, infatti altri compilatori Prolog come YAP Prolog forniscono delle API compatibili con SWI-Prolog, dando la possibilità di realizzare una logica di controllo eseguibile da differenti piattaforme Prolog.

Il sistema centrale ha l'obiettivo di analizzare lo stato del mondo, ricevuto come input sotto forma di informazione sensoriale – valutate tramite unità di misura definite, variabili in base alle componenti utilizzate ed alle preferenze dell'utente (ad es. l'utente può scegliere se ricevere le informazioni di temperatura in gradi Celsius, Kelvin o Fahrenheit) – ricavati dai sensori collegati direttamente tramite le GPIO o indirettamente tramite MCU esterne (a loro volta connesse alla centrale tramite l'ausilio di protocolli di comunicazione), confrontarlo con uno stato obiettivo (le cui caratteristiche sono già disponibili o fornite in input dall'utente tramite una GUI o altro) così da pianificare una serie di operazioni valide per il raggiungimento dell'obiettivo attraverso l'interazione tramite degli attuatori, collegati a degli oggetti presenti nel mondo e capaci di modificarne lo stato nel modo desiderato.

Gli oggetti vengono “pilotati” dalle periferiche – tramite appositi attuatori – in direzione dello stato obiettivo desiderato, e la buona riuscita delle azioni a loro ordinate viene verificata dal sistema tramite un sistema di feedback; in alcuni casi, tale sistema è implementato tramite appositi sensori presenti nella stessa MCU, dedicati alla verifica diretta dello stato dell’oggetto, in altri casi, sfruttando i sensori già presenti nel mondo su altre MCU dedicate al monitoraggio del parametro che dovrebbe modificarsi, tramite un controllo dei suoi valori prima e dopo l’esecuzione dell’azione.

Quest’ultimo è il caso meno efficiente e sotto alcuni aspetti, anche il meno preciso; preso come esempio la gestione di una lampada, un nodo dedicato al suo controllo completo dotato di attuatori e sensori dedicati come quello di corrente o di luminosità (direttamente puntato verso la lampada) riesce a dare un feedback preciso, immediato e sicuramente affidabile, soprattutto se confrontato con i feedback restituiti nodo dotato di sensori di luminosità dedicati al controllo generale del livelli di luce nella stanza, infatti, a seconda della luce già presente all’interno della stanza, insieme al fattore potenza della lampada ed il grado di sensibilità del sensore, non è sempre detto che vengano rilevate delle differenze tangibili tra i valori rilevati nell’ambiente prima e dopo l’esecuzione del comando.

5 Componenti secondarie

Le componenti secondarie del sistema distribuito possono essere di varia natura, ma tutte hanno in comune la capacità di interfacciarsi con il sistema centrale tramite un protocollo di comunicazione sicuro e la capacità di interagire e/o raccogliere informazioni dall'ambiente.

Per il ruolo di componenti secondarie sono state prese in considerazione due diversi dispositivi, il Raspberry Pi – già citato nel ruolo di unità centrale – ed il micro controllore ESP8266 della Espressif Systems, nelle due varianti 12-E e 12-F; questa MCU a basso costo risalta per la presenza di un chip Wi-Fi (IEEE 802.11 b/g/n) con pieno supporto al protocollo TCP/IP grazie ad uno stack di istruzioni già presente al suo interno, oltre che consumi contenuti (sempre se vi è una corretta gestione del modulo Wi-Fi). Questo tipo di MCU, vista le sue caratteristiche è per sua natura particolarmente adatta per essere applicata nel contesto dell'IoT, di fatto, benché possano essere implementati svariati protocolli di comunicazione, in questa attività progettuale si è fornito supporto al solo TCP/IP.

Fig. 8 ESP-8266 12F

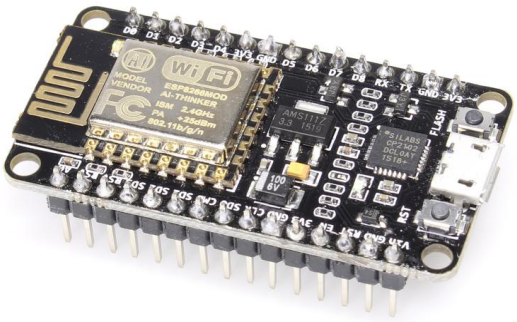


Fig. 9 ESP-8266 12E Dev Board AMICA

La MCU ESP-8266, vanta una CPU 32-bit con architettura RISC della Tensilica Xtensa con frequenze pari a 80 MHz (con possibilità di incremento fino a 160MHz, con stabilità variabile da chip a chip), 64 KiB di RAM per le istruzioni, 96 KiB di RAM per i dati, memoria flash di 4 MiB e 16 pin GPIO ed un ampio parco SDK che permettono lo sviluppo in diversi linguaggi di programmazione, alcune delle quali particolarmente mirate al contesto d'uso dell'Internet delle Cose.

Per la seguente progettualità sono state utilizzate due varianti distinte della ESP8266; la prima variante è la 12F stand-alone, mentre la seconda variante è la devBoard NodeMCU Amica che gode della presenza del modulo 12E già accoppiato con un Bridge USB-to-UART CP2102 per una scrittura della memoria flash rapida ed immediata. Tra le due varianti di ESP8266 utilizzate durante la progettazione del sistema non ci sono differenze sostanziali, se non per la presenza delle approvazioni FFC e CE ed la potenza dell'antenna che secondo documentazione risulta migliorata nella variante 12F, tuttavia per quanto riguarda la disposizione dei pin, così come tutte le relative funzionalità, le due versioni risultano equivalenti, il che ha facilitato la creazione della base di conoscenza in questione, in quanto è bastato inserire le regole in grado di descrivere le informazioni di una sola delle due varianti della MCU perché queste risultassero valide per entrambe in fase di generazioni delle istruzioni FORTH.

All'interno di esse si è provveduto all'inserimento di un interprete FORTH (nello specifico si è optato per lo sviluppo di una versione basata su PunyForth pesantemente modificata per andare incontro alle necessità progettuali), utile ad interfacciare agevolmente la MCU ai vari sensori e attuatori collegati alle porte GPIO, secondo dei protocolli definiti che variano a seconda della periferica collegata. Oltre ad essere un potente strumento per la programmazione efficiente dell'hardware – evitando le difficoltà di codifica tipiche di Assembly [19, 20] – la scelta di FORTH è dovuta anche ad altri aspetti come la possibilità di compilare sequenze di comandi per una loro successiva esecuzione – mettendo a disposizione un vero e proprio compilatore portatile – come pure la possibilità di essere eseguito in modalità interattiva come fosse una shell di comando [21, 22].

È tuttavia necessario fare alcune precisazioni; come nel caso di Prolog con i suoi interpreti, anche per quanto riguarda FORTH esistono diverse implementazioni ognuna con le sue caratteristiche. Volendo concentrare il discorso sulle sole implementazioni in esame, quindi il gForth usato nell'unità centrale e PunyForth usato nelle ESP8266, già tra sole queste due implementazioni è possibile identificare alcune differenze che è necessario vengano tenute in considerazione se si vuole rendere operativo il sistema (che il codice generato dall'agente intelligente sia corretto e non generi errori).

Volendo fare alcuni esempi in merito, alcune istruzioni presenti in un dato interprete sono assenti nell'altro, esistono casi in cui la "stessa" istruzione è richiamabile dai due interpreti attraverso istruzioni leggermente differenti, alcune istruzioni che teoricamente dovrebbero essere equivalenti in verità non operano allo stesso modo (in alcuni casi il processo per ottenere il risultato è importante tanto quanto il risultato stesso), in più PunyForth, a differenza di gForth, è case sensitive, e così via.

Per porre rimedio al problema su menzionato, è stato necessario fare una scelta e più nello specifico, si è dovuto decidere quale doveva essere la sintassi di riferimento da utilizzare per l'intero progetto; in questa attività di tesi, si è deciso di prendere la sintassi gForth come riferimento principale nella generazione del simbolismo necessario. Nel caso specifico di PunyForth si è creato un layer di compatibilità composto da dichiarazioni, scritte nella sintassi PunyForth con l'obiettivo di tradurre alcune istruzioni gForth nel corrispettivo valido in PunyForth, ospitabile sia in memoria flash all'interno della MCU che all'interno della base di conoscenza (in tal caso l'agente intelligente si occupa di inviarle alla MCU come parte della sua configurazione), in questo modo l'agente intelligente ha la possibilità di generare il codice nella sola sintassi standard gForth, senza doversi preoccupare delle diverse varianti esistenti. In caso di ulteriori MCU con il loro interprete FORTH, è necessario risolvere eventuali differenze ed incompatibilità allo stesso modo già descritto in precedenza.

Come modalità di collegamento, considerata a scopo progettuale in fase di sperimentazione, tra le varie MCU dedicate al ruolo di componenti secondarie ed unità centrale si è concentrata maggiore attenzione sull'uso di una configurazione di una rete ad-oc Wi-Fi (2.4Ghz) con meccanismi di sicurezza WPA2; tutte le componenti del sistema si sono considerate come associate alla stessa subnet di rete. Idealmente, l'intero sistema distribuito considerato durante lo svolgimento dell'attività di progettazione presenta una struttura simile a quella illustrata nell'immagine 9; questa struttura, specie considerando l'uso del meccanismo di sicurezza WPA2 si potrebbe considerare sicura nei confronti di agenti esterni, tuttavia tale meccanismo potrebbe risultare insufficiente nei confronti di eventuali altri nodi di rete presenti nella stessa subnet, non direttamente correlati al corretto funzionamento del sistema, casistica che a seconda del contesto d'uso non è affatto scontata e per di più inaccettabile.

Si noti che, allo stato dell'arte, l'intera architettura generale prevede che le comunicazioni possano essere inizializzate solo dalla componente centrale e non dai nodi periferici che di fatto si limitano a reagire come indicato, eseguendo i comandi inviati dall'unità centrale e restituendo delle risposte, il tutto solo quando richiesto dall'agente intelligente. A questo si lega l'ulteriore svantaggio da parte dei nodi di dover mantenere sempre attiva la connessione Wi-Fi, con ovvio dispendio energetico.

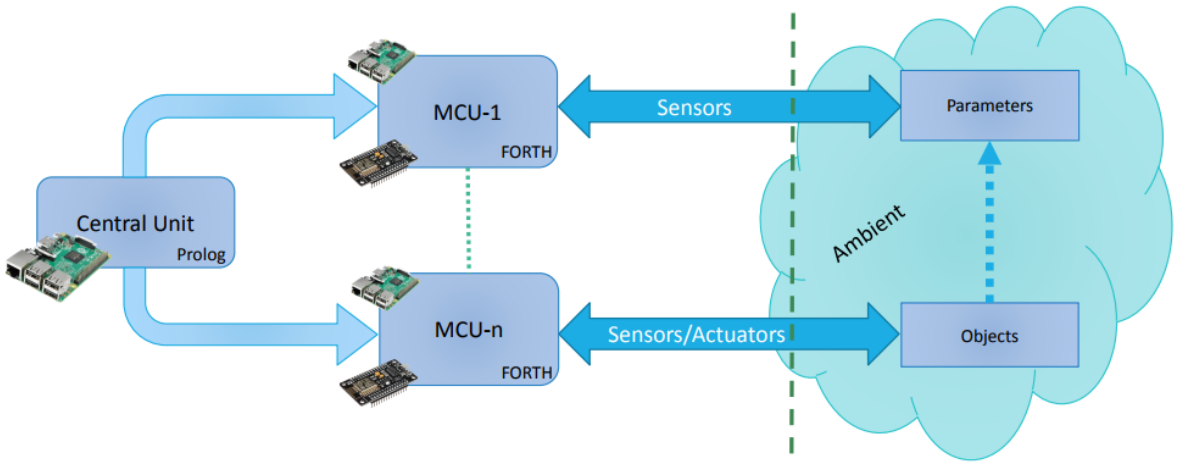


Fig. 10 Struttura del sistema distribuito

6 Interazione sicura tra componenti

La parte del sistema di regole dedicata alla comunicazione tra i nodi, procede prima all'analisi delle specifiche del dispositivo interessato, così da ottenere una configurazione valida allo scopo, basando tale analisi su informazioni già presenti all'interno della base di conoscenza in quanto inserite manualmente da parte dell'utente, così da comunicare ai dispositivi la configurazione ed i vari comandi da eseguire in produzione. Tale possibilità di gestione e configurazione che di fatto costituisce il supporto ad una data MCU, è garantito a seguito del censimento manuale di tutte le sue informazioni – numero di pin e tutte le relative funzionalità supportate – in quanto il sistema non è in grado di ricavare tali informazioni autonomamente.

Vista l'ampia gamma di oggetti che il sistema ha la potenziale capacità di gestire (essi spaziano dalla semplice lampadina fino a arrivare a interi sistemi dedicati alla sicurezza) è imperativo che esso garantisca all'utente finale che l'interazione tra le componenti del sistema avvenga secondo modalità sicure e senza intoppi: di fatti un'utente non autorizzato non deve avere la possibilità di accedere ad informazioni sensoriali, o peggio, gestire direttamente i vari oggetti e dispositivi (tramite le MCU dotate di attuatori) inviando direttamente i comandi.

Nel protocollo di comunicazione, anche ai fini di garantire la possibilità di svolgere delle sessioni di debugging, è presente una modalità di invio non sicuro dei messaggi – ovvero non criptata – tra unità centrale e i nodi del sistema che l'utente è libero di attivare o meno.

6.1 Canale sicuro

Le informazioni trasmesse possono transitare in canali sicuri da eventuali manomissioni da parte di agenti esterni, grazie all'implementazione di un algoritmo di cifratura coadiuvato da uno di codifica; a scelta l'utente può scegliere tra gli algoritmi di cifratura DES-CBC o AES128-CBC mentre per la codifica viene utilizzato la Base64; nulla impedisce l'implementazione di ulteriori protocolli di sicurezza oltre quelli già definiti come ad esempio algoritmi di cifratura che generano stringhe cifrate composte da soli caratteri stampabili.

La scelta di procedere alla codifica delle stringhe – oltre alla loro cifratura – è stata presa dal momento che all'interno delle stringhe generate dal solo processo di cifratura, svolti tramite DES e AES128, potevano essere presenti dei caratteri non stampabili in grado di dare origine ad errori di interpretazione da parte delle MCU; infatti, il server REPL presente all'interno di ogni MCU (interpellabile sia tramite l'unità centrale che direttamente dall'utente per mezzo di connessione seriale) che esegue le stringhe simboliche inviate dall'unità centrale, alla presenza di determinati caratteri "fuori posto" (come ad es., "*NUL*", "*CR*" ed "*LF*") normalmente vengono utilizzati come token di riconoscimento da parte del parser della MCU può generare degli errori dovuti all'interruzione prematura della lettura della stringa.

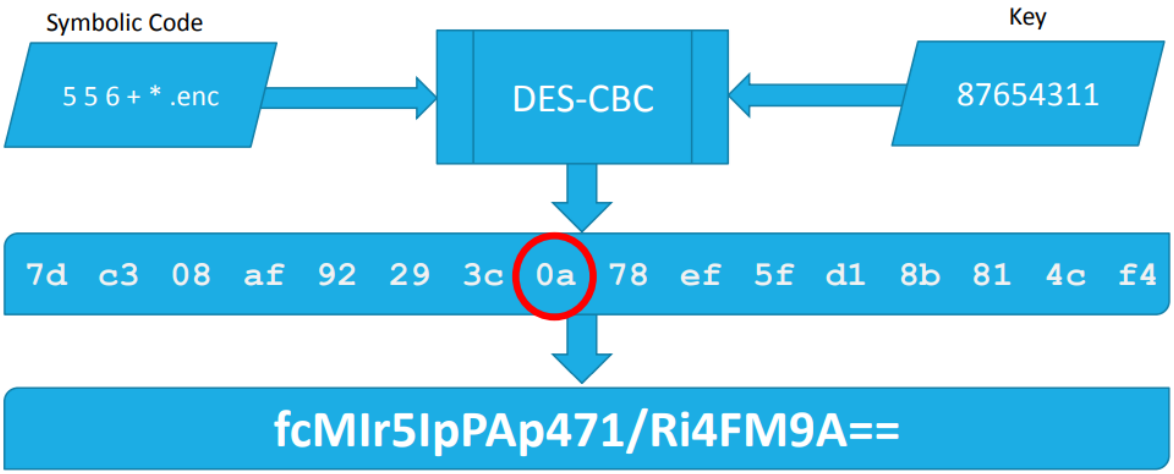


Fig. 11 Codifica della stringa cifrata

All’interno di tutte le MCU è già presente nella memoria flash una MasterKey (da rendere nota solo al diretto possessore/amministratore del sistema), da inserire manualmente all’interno della base di conoscenza dell’agente intelligente, utilizzata in fase di legame tra i dispositivi; il protocollo prevede l’uso di una trasmissione basata su chiavi master simmetriche (unica per ogni MCU) per lo scambio di chiavi di sessione, anch’esse simmetriche, utilizzate nelle effettive comunicazioni. In sostanza il protocollo fa uso di una trasmissione basata su chiavi master simmetriche per MCU per lo scambio di chiavi di sessione utilizzate nelle effettive comunicazioni. L’uso di protocolli a chiavi asimmetriche (es. RSA) per lo scambio di chiavi simmetriche di sessione, sarebbe stata una scelta sicuramente migliore in termini di sicurezza, ma in fase sperimentale, imbattuti nei limiti intrinseci delle varie MCU (in termini di spazio in memoria che di prestazioni) non è stato possibile procedere con la sua implementazione optando per un protocollo più semplice e veloce.

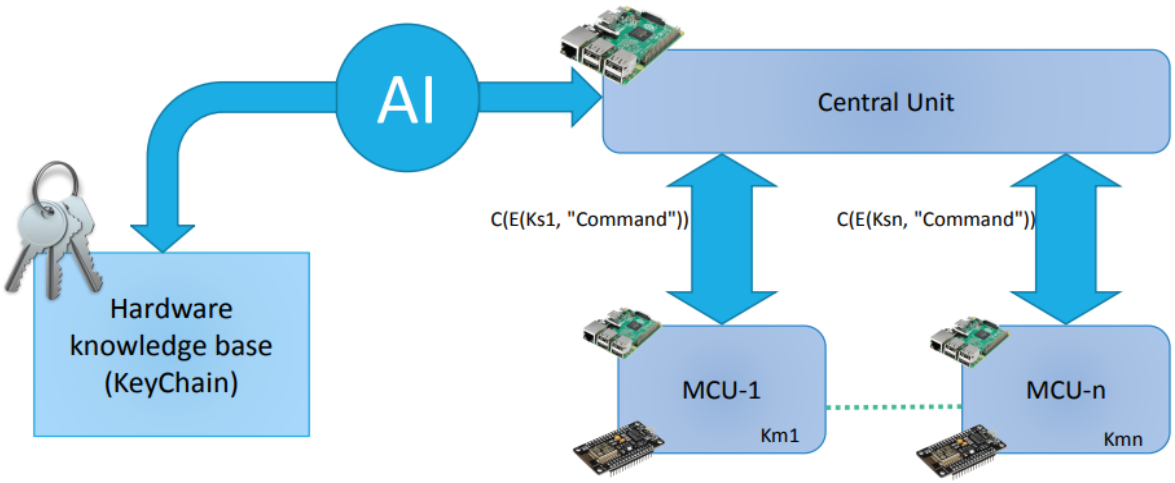


Fig. 12 Comunicazioni tramite canale sicuro

Ogni comunicazione da parte dell’unità centrale con i vari nodi è autenticata dal fatto che ogni chiave è specifica per MCU; l’agente intelligente ha la certezza di comunicare con la specifica MCU in quanto quest’ultima è considerata l’unica in grado di decifrare, interpretare e reagire correttamente al messaggio inviato, allo stesso modo il nodo può considerare i messaggi ricevuti come autenticati in quanto suppone l’agente intelligente come unico a conoscenza della sua chiave.

Nel caso delle MCU, una decifratura non corretta (causato da una cifratura del messaggio con una chiave errata) andrebbe a generare del codice invalido dando origine ad errori di sintassi da parte dell'interprete FORTH, mentre nel caso dell'AI, un controllo del tipo di dato e del contenuto del messaggio restituirebbe come risultato il caso FALSE, poiché messo al corrente di cosa aspettarsi come possibile risposta da parte delle varie MCU.

Nella seguente implementazione, l'unicità delle chiavi (sia della MasterKey che della SessionKey) è dunque il requisito fondamentale su cui si basa l'intero servizio di autenticazione (sarà cura del sistema di regole quello di garantire l'unicità delle SessionKey generate per le varie MCU); nulla impedisce l'inclusione di una componente "nonce", o più semplicemente dei codici di sequenza, all'interno dei messaggi scambiati per garantire la buona riuscita di tale servizio; nel caso si considerasse l'eventuale sviluppo di una componente broadcast per l'invio dei comandi da parte dell'unità centrale, l'uso di una delle due alternative sopra indicate risulterebbe essere non più facoltativa (ulteriori approfondimenti in merito sono presenti nel Cap. 7 "Conclusioni e possibili sviluppi").

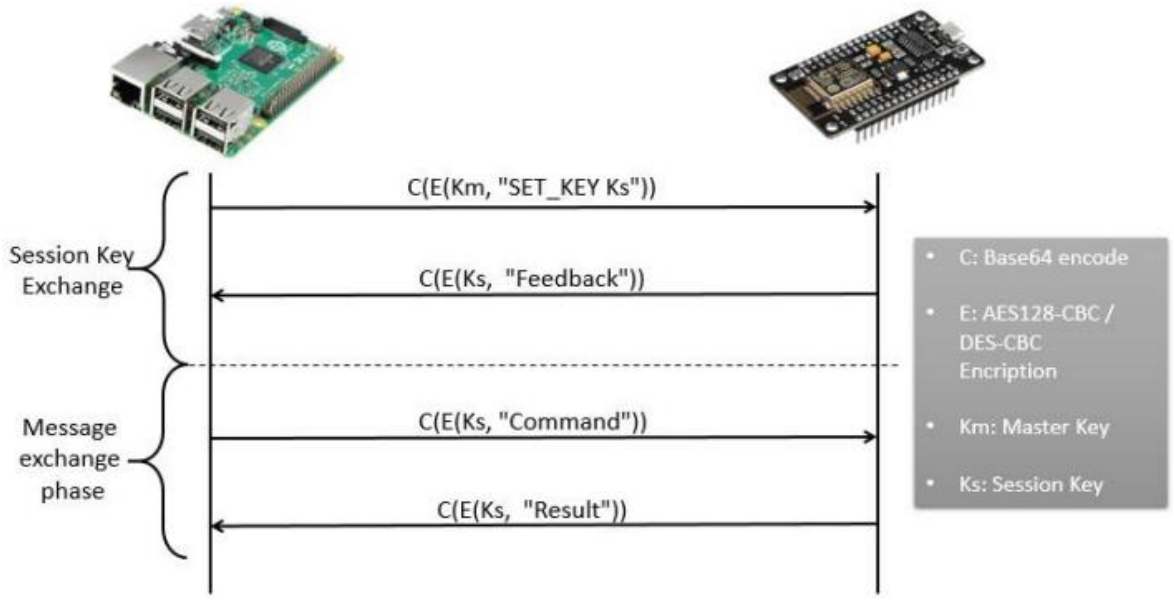


Fig. 13 Procedura di scambio delle chiavi di sessione

Il processo di generazione e scambio delle chiavi di sessione tra l'unità centrale e i vari nodi del sistema, già illustrato in figura 10, è caratterizzato dai tre passi descritti di seguito:

1) L'utente amministratore – che si suppone ente fidato e l'unico a conoscenza della MasterKey della MCU (K_m) – provvede ad inserire manualmente la chiave all'interno della base di conoscenza (attualmente vengono memorizzate in chiaro all'interno della base di conoscenza, senza l'adozione di particolari meccanismi di sicurezza).
2) L'agente intelligente fa inizialmente uso della MasterKey per la cifratura di tutti i comandi indirizzati alla MCU, tuttavia questa accetta solo lo specifico comando "*SET_KEY*" utile per il cambio della chiave, per questo motivo, l'agente intelligente deve prima provvedere alla generazione di una nuova "chiave di sessione" (K_s) (pseudo)random di 8 o 16 caratteri – a seconda dell'algoritmo di cifratura scelto – e di inviarla alla MCU, cifrata tramite la MasterKey.
3) Alla ricezione del comando "*SET_KEY*", sia la MCU che l'agente intelligente salvano la chiave di sessione così che tutti i futuri messaggi vengano cifrati con la suddetta chiave, il tutto allo scopo di limitare il più possibile l'uso della MasterKey a chiaro favore della sicurezza.

La MCU appena avviata, sempre se nei suoi settaggi è previsto l'uso della crittografia per i messaggi ricevuti ed inviati, entra in una sorta di modalità con cui non gli è negata la possibilità da eseguire dei comandi, tutto ciò fino a quando rileva che la MasterKey è ancora la chiave di crittografia in uso. L'obiettivo è quello di forzare il sistema di regole, così come un qualsiasi altro utilizzatore, a settare una nuova chiave temporanea usando il meno possibile la MasterKey presente all'interno della memoria flash; è infatti noto che l'utilizzo intensivo della stessa chiave di cifratura espone questa a maggior pericolo di rilevamento a seguito di attacchi di criptoanalisi (tale pericolo si fa più concreto all'aumentare della dimensione dei messaggi inviati e ricevuti).

La chiave di sessione, in quanto tale, dispone di un periodo di validità limitato, la cui variabilità può dipendere da svariati aspetti; più nello specifico, tra i casi che possono portare alla scadenza della chiave di sessione è possibile distinguere i seguenti:

1) Rinnovo: l'invio automatico (tramite processo cronicizzato all'interno del sistema di regole) o manuale (direttamente da parte dell'utente) di un nuovo comando "*SET_KEY*" con allegata una nuova chiave di sessione, il tutto cifrato tramite la chiave di sessione valida fino a quel momento, porta alla sostituzione della precedente chiave.
2) Desincronizzazione: lo spegnimento della MCU (dovuto per esempio dall'interruzione dell'alimentazione o altro), il reset (causato da un possibile errore di sistema) porta alla perdita della chiave di sessione da parte della MCU che di fatto la teneva memorizzata nella RAM; ciò richiede, da parte dell'agente intelligente, l'invio di una nuova chiave di sessione tramite lo stesso comando "*SET_KEY*" ma cifrato nuovamente con la MasterKey, ignorando a tutti gli effetti la chiave di sessione in suo possesso in quel momento. Effettuata la re-sincronizzazione della chiave di sessione le comunicazioni tra unità di controllo e nodi riprenderanno come di consueto.

La scelta di utilizzare le chiavi di sessione trova motivazione nella volontà di rendere più difficoltosa la violazione del sistema in quanto lo sforzo effettuato per l'ottenimento della chiave, tramite un attacco di tipo brute-force, verrebbe facilmente reso vano da un semplice cambio della chiave di sessione, inoltre limitare l'uso della MasterKey al solo scambio iniziale della chiave di sessione fornisce una quantità di informazione minima utile, ritenuta nella maggior parte dei casi insufficiente per una eventuale cripto-analisi.

La capacità di resistenza ad attacchi di tipo cripto-analisi è dipendente dalla natura dell'algoritmo di cifratura scelto, nello specifico tra DES-CBC e AES128-CBC; è infatti ben noto che la cifratura AES128-CBC è più efficace se paragonata a DES-CBC che di contro risulta essere più rapida da eseguire (aspetto da non sottovalutare, soprattutto viste le risorse hardware molto limitate che caratterizzano i dispositivi embedded che se ne devono fare carico, utilizzate nella seguente attività progettuale).

6.2 Trasmissione del codice

In base alla MCU disponibile (o specificata) e i vari sensori e attuatori che si desidera collegare a quest'ultima, l'agente intelligente effettua la generazione della configurazione (quindi espande la base di conoscenza con nuovi fatti) tramite lo svolgimento dei seguenti passi:

1) Attraverso delle regole e dei fatti già disponibili all'interno della base di conoscenza, analizza le varie porte GPIO necessarie per i collegamenti dei sensori e/o attuatori (considerandone le shield in uso) e quelle disponibili da parte delle MCU.
2) Cerca una possibile configurazione valida, e in caso di successo, arricchisce la base di conoscenza con dei fatti che riescono a rappresentare la nuova configurazione valida trovata.
3) Avendo a disposizione i nuovi fatti, l'agente intelligente è in grado di generare il codice FORTH (configurazione ed istruzioni) utile alla corretta esecuzione dei vari comandi da parte della MCU, ai fini di rendere possibile il raggiungimento dell'obiettivo desiderato dall'utente.
4) Il codice generato per la particolare MCU viene quindi cifrato tramite l'ausilio di uno dei due algoritmi di cifratura implementati (AES128-CBC o DES-CBC) e codificato in Base64 così da risolvere eventuali problemi interpretativi, ed infine inviato tramite un protocollo di comunicazione (TCP/IP o Socket UNIX).

Per ottenere tale risultato è stato necessario rendere note all'agente intelligente non solo le varie MCU i sensori e gli attuatori, ma anche le relative GPIO di cui sono composte con tutte le loro rispettive funzionalità.

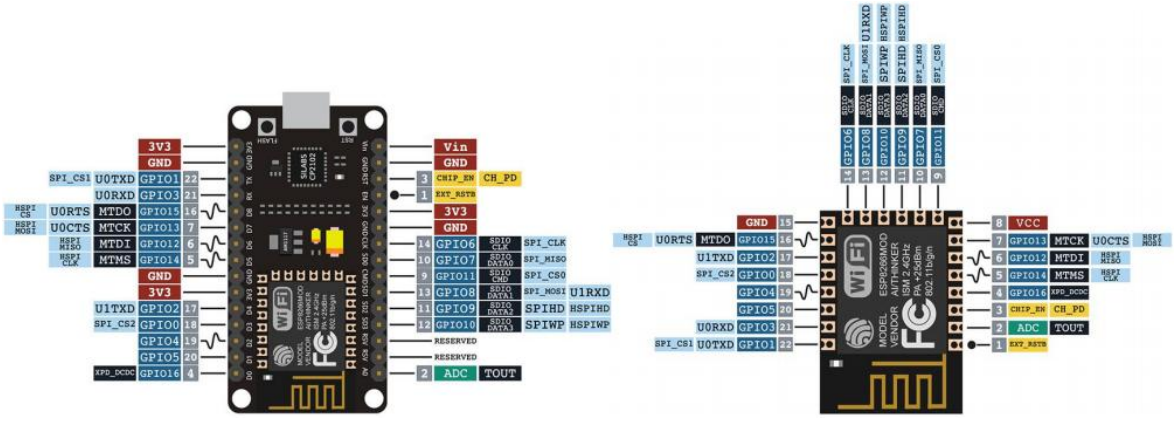


Fig. 14 Modalità di funzionamento dei Pin

Il sistema ha il compito di generare il codice secondo un formalismo generale ed indipendente dalla specifica MCU o configurazione basato su un formalismo simile alla sintassi del linguaggio FORTH; tutto ciò di fatto rende possibile all'utente finale, l'interazione ed il controllo completo del nodo dei vari dispositivi ad esso collegati in modo trasparente, poiché il set di istruzioni generato è a tutti gli effetti indipendente dalla specifica MCU e dai collegamenti dei vari PIN presenti indicati nella configurazione. Più precisamente le operazioni messe a disposizione dal sistema, dipendono dai nodi e (più ad alto livello) dagli oggetti ad essi collegati e/o parametri ambientali monitorati.

Oltre le informazioni riguardo alla MCU – la lista dei PIN e le relative funzionalità supportate – presenti all'interno della base di conoscenza del sistema, il sistema di regole è in grado di amministrare ulteriori informazioni; nello specifico si tratta di file sorgenti FORTH, contenenti una lista di precise istruzioni e dichiarazioni nel suddetto linguaggio, fondamentali al corretto supporto delle istruzioni generate dal sistema di regole. Come già descritto nel capitolo 5 di questo elaborato, "Componenti secondarie", l'ambiente FORTH può differire per alcuni aspetti dall'implementazione standard – si ricordi che in questa progettualità si è deciso di realizzare un sistema di regole in grado di generare il codice FORTH necessario seguendo come base di riferimento la sintassi dell'interprete gForth – e che per questo motivo è necessario "adattare" il codice simbolico generato alle variazioni presenti.

In fase di configurazione, è compito dell'agente intelligente (dopo aver settato la chiave di sessione) quello di leggere e trasferire le informazioni contenute nel file, istruzioni e dichiarazioni FORTH che la MCU memorizza nella RAM. Invece di utilizzare dei file statici – pratica che si è comunque dimostrata efficace, visto anche il limitato numero di varianti dell'interprete FORTH prese in esame – sarebbe stato possibile memorizzare nella base di conoscenza le informazioni relative ai vari interpreti, così da implementare una componente capace generare dinamicamente il layer di compatibilità necessario (si lascia questa ulteriore idea come possibile sviluppo futuro di questo progetto).

7 Tipologie di sensori e attuatori

Vista l'ampia gamma di tipologie di sensori e attuatori disponibili nel mercato, si è sfruttata la possibilità di creare diverse configurazioni di nodi costituite da una MCU e una o più componenti ad essa collegate. Il nodo risultante, classificabile con il termine "black-box", sono in grado di avere una interazione attiva (tramite degli attuatori collegati agli oggetti) o passiva (sola presenza di sensori utili al monitoraggio dello stato) con l'ambiente. La nomea di scatola nera sta a sottolineare la trasparenza con cui il sistema è in grado di garantire tali componenti nei confronti dell'utente, giustificata dal fatto che esso non ha necessità di sapere la composizione interna di queste configurazioni in quanto essa è interamente amministrata dalla logica artificiale; l'utente finale può usufruire di una lista di funzioni ad alto livello che dipende strettamente da tutte le black-box presenti nel mondo.

L'utente sviluppatore ha la possibilità di creare le proprie black-box specificando la MCU e le periferiche che si desidera collegare ad essa, la cui effettiva possibilità di collegamento e generazione di una configurazione valida (sia dal piano hardware che software) dipende dalle capacità e disponibilità di porte GPIO della MCU oltre che dai requisiti dei singoli sensori e attuatori che si desidera collegare – aspetti di cui si occupa interamente la logica artificiale –.

Attraverso delle apposite query, è possibile ottenere da parte dell'agente intelligente tutte le possibili configurazioni per ogni MCU, specificando solo i parametri che si desidera monitorare e quali oggetti amministrare (è anche possibile specificare le MCU da tenere in considerazione durante il processo); l'utente deve quindi procedere alla realizzazione fisica dei collegamenti indicati di tutte le componenti mentre il sistema di regole provvede alla connessione ed alla configurazione della base di conoscenza sulla base di esse (oltre che al comando).

7.1 I Sensori/Attuatori

Tra i sensori considerati per l'attività di progetto si distinguono: sensore di corrente ACS712, di luminosità GL5516, potenziometro digitale, sensore di Movimento HC SR501, sensori di gas o altre sostanze nell'aria (MQx, con x tra 1 e 12, a seconda delle tipologie di gas che è capace di rilevati), temperatura e umidità (DHT11 e DHT22 a seconda del range, delle tolleranze e dei livelli di sensibilità del sensore), sola temperatura DS18B20 ed infrarossi, mentre tra gli attuatori utili all'attività di progetto si distinguono: relay, trasmettitore IrDA, led (sia RGB che monocromatici) e buzzer (sia attivo che passivo). Sia sensori che attuatori sono da considerare comprensivi di shield, quindi già pronte per il collegamento alle MCU. Alcuni esempi sono visualizzabili nell'immagine sottostante.

Fig. 15 Alcuni sensori e attuatori utilizzati

Per ognuna delle seguenti componenti sono state censite all'interno della base di conoscenza dell'agente intelligente, tutte le caratteristiche dei vari shield (si è scelto il seguente livello come base di descrizione minima dei sensori e attuatori, ma sarebbe stato possibile scendere a livelli descrittivi più bassi) tra cui: parametri misurati (nel caso di sensori), unità di misura, numero di pin con le relative funzionalità, tolleranze di errore e direzione dei protocolli di comunicazione (input/output).

7.2 Le black-box

Tra le black-box considerate per l'attività di progetto si possono distinguere le seguenti configurazioni:

1) Alimentazione: capacita di gestione dell'alimentazione basico di dispositivi elettronici, in particolar modo, quelli privi di logica ad alto livello interna; essa consiste di uno o più relay e relativi sensori di corrente usati come meccanismo di feedback.
2) Alimentazione 2: si tratta di una black-box di alimentazione basica con l'aggiunta di un potenziometro digitale, capace di gestire la quantità di corrente da fornire agli oggetti, particolarmente indicata per le lampade di cui si vogliono regolare i livelli di luminosità.
3) Gestione luminosità: black-box ideata per la gestione della luminosità dell'ambiente; un sensore di luminosità permette di rilevare la quantità di luce nell'ambiente, permettendo una gestione più attenta dei dispositivi di luminosità tenendo in considerazione l'effettiva necessità di una variazione di tale parametro ambientale; essa consta di relay, sensore di corrente, sensore di luminosità.
4) Gestione luminosità 2: si tratta di una black-box di gestione di luminosità con l'aggiunta di un sensore di movimento che, oltre ai parametri sopra citati, verifichi l'effettiva presenza di utenti nell'ambiente; essa è composta da relay, sensore di corrente, sensore di movimento, sensore di luminosità.
5) Trasmettitore IrDA: solo trasmettitore IrDA, utile attuatore capace di interfacciarsi in modo non invasivo con dispositivi come televisori, condizionatori o altri dispositivi dotati di fabbrica di ricevitori IrDA.

Si ribadisce che l'utente è libero di specificare la propria configurazione desiderata e che non è assolutamente vincolato all'uso delle sole sopra citate.

Il controllo di queste configurazioni può avvenire tramite interfacciamento diretto (comandi diretti dell'utente) o pianificazione (azione dell'agente intelligente eventualmente pilotata dai parametri specificati dall'utente).

8 L'implementazione

All'interno dei sorgenti del programma Prolog sono presenti diversi fatti e regole sotto forma di clausole di Horn che verranno qui presentati in base al contesto d'uso (BOARD, SYSTEM, APPLICATION); per ognuno di essi esistono nel progetto dei file dedicati contenenti fatti e regole con relative arità, le quali caratteristiche sono descritte nei paragrafi che seguono.

8.1 BOARD

Per ogni sensore, attuatore e MCU usati durante i test applicativi, attraverso l'uso di fatti, è stata effettuata una descrizione a livello di BOARD dei PIN disponibili e delle relative funzionalità usando, dove possibile, le nomenclature standard documentate sui datasheet, così da rendere il tutto più immediato (specie a l'occhio dell'utente esperto con questa tipologia di dispositivi).

8.1.1 pin.pl

All'interno del file "pin.pl", si sono definite tutte le caratteristiche generali utili alla rappresentazione del concetto di PIN e le relative caratteristiche che esso può avere, tra le quali si distinguono:

1) Funzioni: ovvero tutte le modalità d'uso che è previsto possa avere il PIN secondo progetto che può essere un'unica funzione o, in alcuni casi, anche più di una. In ogni caso, durante la configurazione è possibile impostarne solo una alla volta. tra le funzioni si distinguono "*vcc"*, "*gnd"*, "*digital_input"*, "*digital_output"*, "*sda"*, "*scl"*, "*analog_input"* ed "*analog_output"*.
2) Relazioni tra le funzioni: preso un esempio pratico, definisce che un PIN con funzione "*digital_input*" può essere collegato ad un altro PIN con funzione "*digital_output*", o che i pin con funzioni "*vcc*" e "*gnd*" possono essere collegati a loro volta ad altri pin con funzioni equivalenti "*vcc*", "*gnd*" e così via.

3) Simboli utilizzati nel linguaggio simbolico FORTH eseguibile all'interno delle MCU, associati alle funzioni dei PIN: "*GPIO_IN*", "*GPIO_OUT*", "*GPIO_OUT_OPEN_DRAIN*".

Tra le regole utilizzate per la descrizione di questo contesto è possibile distinguere le seguenti:

- `pin_function_type/1`: definisce una funzione disponibile su uno specifico PIN della MCU.
- `pin_function_mode/2`: definisce una specifica sequenza simbolica da eseguire sulle MCU con quel determinato PIN.
- `pin_function_relation/2`: definisce le relazioni tra le funzioni dei PIN, come già descritto in precedenza.
- `pin_function_type_skip/1`: ha l'obiettivo di aggiunge un'eccezione durante il processo di controllo sulla disponibilità dei PIN.

8.1.2 peripheral.pl

All'interno di questo secondo file si trovano i sorgenti relativi alla descrizione di fatti e le regole utili alla definizione delle varie tipologie di periferiche trattate (ovvero dei sensori e degli attuatori), comprese le loro caratteristiche distintive, tra cui:

1) Tipologia: definisce il particolare sensore o attuatore in base al tipo.
2) Classe: definisce la classe di periferica (es: relay, led, buzzer, dht) utile a raggruppare le tipologie di periferiche con caratteristiche più o meno comuni.
3) PIN: nello specifico quanti e quali PIN sono presenti nella shield.
4) Parametri ambientali: rilevabili o controllabili dai sensori e dagli attuatori rispettivamente, a seconda di cosa si sta descrivendo.
5) Funzioni associate ai PIN: da eseguire attraverso l'interprete dell'MCU ("*high*", "*low*", "*analog_read*"), in genere coinvolgono un singolo PIN della Periferica.
6) Funzioni associate alle Periferiche: che interessano contemporaneamente più PIN della periferica ("*on*", "*off*", "*read*").

L’insieme di fatti e regole utilizzati per la descrizione del seguente contesto sono le seguenti:

- `peripheral_class/1`: la seguente regola è utile all’assegnazione di una classe ad un particolare sensore/attuatore.
- `peripheral_type/1`: la seguente regola è utile all’assegnazione di una tipologia ad un particolare sensore/attuatore.
- `peripheral_actuator/1`: la seguente regola assegna la particolare periferica alla macro categoria “attuatori”.
- `peripheral_sensor/1`: la seguente regola assegna la particolare periferica alla macro categoria “sensore”.
- `peripheral_actuator_parameter/2`: definisce quale parametro ambientale è rilevabile o controllabile dal particolare attuatore (per quelli che riescono ad agire direttamente sul parametro dell’ambiente senza l’ausilio di un oggetto; Es. buzzer attivo o passivo, led monocromatici ed RGB, etc.).
- `peripheral_sensor_parameter/2`: definisce quale parametro ambientale è monitorabile dal particolare sensore.
- `peripheral_type_id/2`: tramite questa regola si associa una tipologia di periferica ad un codice identificativo numerico utile in fase di generazione ed esecuzione del codice FORTH.
- `peripheral_type_alias/2`: tramite questa regola si associa una tipologia di periferica ad un alias (una stringa di testo), utile in fase di generazione ed esecuzione del codice FORTH.
- `peripheral_type_class/2`: tramite questa regola si associa l’intera tipologia di periferica ad una classe di appartenenza.
- `peripheral_name/2`: l'istanzia una particolare periferica definendone un nome univoco arbitrario all’interno del sistema.
- `peripheral_object_connection/2`: questa regola Prolog definisce la connessione tra una periferica e un oggetto.
- `peripheral_pin/4`: questo fatto riesce a contenere informazioni sui vari PIN delle diverse periferiche richiedendone il tipo, il numero del PIN, la funzionalità ricoperta, ed una stringa che utilizzata in fase di generazione del codice FORTH.

- `peripheral_pin_set_mode_procedure/2`: prepara la sequenza di simboli utili all'impostazione di una data modalità della GPIO sulla MCU, sempre se il PIN prevede una funzionalità per cui è richiesta l'esecuzione di tale procedura.
- `peripheral_pin_function_definition/3`: in questa regola si va a definire il codice simbolico valido per una determinata funzionalità da eseguire su uno specifico PIN.
- `peripheral_function_prepare_code/2`: questa regola ha il compito di formattare correttamente il codice simbolico da eseguire sulla MCU.
- `peripheral_pin_function/5`: questa regola ha il compito di definire, per la particolare periferica, l'azione da eseguire sullo specifico PIN settato con una specifica modalità.
- `peripheral_exec_function_gen_code/4`: tale regola genera il codice simbolico di una istanza della periferica, per una particolare funzione richiamata. Essa restituisce come risultato il codice simbolico utile allo scopo, previa formattazione da parte della regola già descritta sopra, "`peripheral_function_prepare_code/2`".
- `peripheral_exec_function/2`: per una specifica componente ed una specifica azione da compiere, la seguente regola va a generare il codice simbolico necessario e lo manda in esecuzione contattando direttamente la MCU a cui la periferica interessata (sensore o attuatore) è collegata tramite la "`mcu_send_message/2`".
- `peripheral_exec_function/3`: come nel caso dell'istruzione già descritta sopra, "`peripheral_exec_function/2`", ma che in più prevede la ricezione di una risposta da parte della MCU (un risultato o anche un valore di feedback).
- `peripheral_object_check_usability/2`: durante la fase di associazione di una periferica ad una caratteristica o funzionalità di un determinato oggetto, questa regola controlla se la tipologia di quel particolare oggetto può essere effettivamente associato a quella specifica tipologia di periferica in particolare, all'interno della base di conoscenza sono presenti le informazioni relative agli oggetti che il sistema tratta nel contesto della domotica con le relative funzionalità disponibili.

- `peripheral_object_check_usability/4`: come nel caso della regola già descritta sopra, "`peripheral_object_check_usability/2`", ma che in più prevede lo svolgimento di una procedura di controllo direttamente su istanze di oggetti e periferiche.
- `peripheral_object_make_connection/2`: se i controlli svolti tramite la regola "`peripheral_object_check_usability/4`" danno esito positivo, questa regola associa una periferica e un oggetto tramite l'assert di "`peripheral_object_connection/2`".
- `peripheral_init/1`: a partire da una lista di liste (dove ogni elemento ha il formato ["*PeripheralName"*, *"ObjectName"*]), questa regola consente di associare gli oggetti alle periferiche; in caso di fallimento, questa regola richiama altre regole di rollback utili a ritrattare tutte le informazioni asserite fino a quel momento.

8.1.3 mcu.pl

Anche per le MCU, come per le periferiche, sono state definite tipologie e caratteristiche, definendo i tipi di MCU trattati (pi, esp8266_12e) e la loro descrizione a livello di BOARD (quindi PIN e funzionalità), più una serie di regole per inviare all'interprete della MCU la sequenza simbolica da eseguire, come settare la MasterKey per utile alla comunicazione cifrata ed alla configurazione della MCU.

Di seguito è presente una lista di tutti i fatti e le regole presenti all'interno del suddetto file Prolog:

- `mcu_type/1`: questa regola è utile ad indicare tutte le varie tipologie di MCU trattate dalla logica artificiale.
- `mcu_file/2`: indica la presenza di eventuali file di inizializzazione della MCU utili all'implementazione del layer di compatibilità tra le diverse implementazioni FORTH esistenti; ulteriore descrizione dell'utilità della seguente regola è presente nel capitolo 6, paragrafo 2, di questo elaborato, "Trasmissione del codice".
- `mcu_pin/3`: definisce i PIN presenti sulla BOARD della MCU, descrivendone anche tutte le relative funzionalità supportate.

- `mcu_peripheral_pin_connection/6`: regola che definisce la connessione tra una periferica (sensore o attuatore) con una MCU, dettagliando precisamente quali sono i PIN coinvolti e con quale funzionalità i due dispositivi sono collegati.
- `mcu_net_key/3`: regola che definisce la chiave – più nello specifico la MasterKey – di crittografia usata per garantire la segretezza delle informazioni trasmesse tra il sistema di regole e la MCU durante le comunicazioni necessarie. Allo stato dell'arte questa è memorizzata in chiaro direttamente nei sorgenti Prolog; è possibile applicare alcuni accorgimenti a riguardo utili ad assicurare un certo livello di sicurezza nello storage della seguente informazione, almeno da parte del sistema di regole.
- `mcu_net_address/3`: la seguente regola Prolog risulta utile in fase di assegnazione di tutte le informazioni, indispensabili alla comunicazione di rete; nello specifico, tra i parametri previsti si nota la presenza dell'indirizzo "IP" e una "Porta" (ovvero quella su cui si trova in ascolto il server REPL ospitato nella MCU.
- `mcu_net_get_key/2`: la seguente regola Prolog risulta utile per risalire alla chiave di cifratura/decifratura per la specifica MCU, nello specifico restituisce la chiave di sessione, se presente, altrimenti la Masterkey, se presente, oppure ancora, il valore "none".
- `mcu_net_set_session_key/2`: regola che ha il compito di impostare la chiave di sessione generata, associandola alla MCU specifica.
- `mcu_net_address_assignment/3`: regola che si occupa di compiere l'assert di "`mcu_net_address/3`", così che il sistema di regole possa usufruirne successivamente in fase di esecuzione.
- `mcu_net_set_session_key/2`: regola che si occupa di genera una chiave (pseudo)casuale – ovvero la SessionKey – e di inviarla alla MCU facendo uso delle regole Prolog già descritte in precedenza; più nello specifico, porta avanti la generazione della chiave di sessione attraverso la regola Prolog "`mcu_send_message/2`", quindi effettua l'assert di questa tramite la regola "`mcu_net_key/3`" così da assegnare la chiave alla MCU.
- `mcu_net_get_key/2`: per una particolare MCU, questa regola restituisce la chiave di cifratura attualmente in vigore (SessionKey o MasterKey), oppure

"none" se la MCU in quel momento non prevede l'uso un algoritmo di crittografia per le comunicazioni.

- `mcu_send_message/2`: Regola utile all'invio di una lista di stringhe simboliche ad una particolare MCU così da farle eseguire all'interprete in esso immagazzinato; si tratta di una delle istruzioni fondamentali con cui si dà al sistema di regole la possibilità di comunicare con tutti i vari nodi a disposizione del sistema.
- `mcu_send_message/3`: regola analoga a quella vista in precedenza, la "`mcu_send_message/2`", ma a differenza di questa si aspetta anche risposta da parte della MCU; come già descritto al termine del capitolo 3, "Sistema di regole e la base di conoscenza", la risposta ricevuta dalla MCU risulta essere una lista di dimensione pari alla lista delle istruzioni inviate originariamente dal sistema di regole, dove a ogni posizione del vettore di ritorno corrisponde (se prevista) la corrispondente risposta all'istruzione inviata.
- `mcu_peripheral_make_connection/6`: a partire dalle informazioni quali, nome della MCU e del dispositivo, PIN e funzionalità, controlla se la connessione non è già esistente, ed in caso, dopo un controllo della disponibilità, la inserisce all'interno della base di conoscenza.
- `mcu_init/1`: a partire da una lista di liste, dove ogni suo elemento (una lista) ha il seguente formato: ["*McuName", "McuPin", "McuPinFunction", "PeripheralName", "PeripheralPin", "PeripheralPinFunction"*]), questa regola consente di associare le varie periferiche alla MCU richiamando la regola Prolog già definita "`mcu_peripheral_make_connection/6`", nel caso di fallimento ritratta tutte le informazioni date tramite apposita regola; ulteriori informazioni sono disponibili nel capitolo 3, paragrafo 2 di questo elaborato ("hardware").
- `mcu_peripheral_check_avaiability/6`: in fase di configurazione di una MCU, questa regola Prolog ha il compito di verificare la disponibilità dei PIN necessari al collegamento sia da parte della MCU che da parte del sensore/attuatore.

- `mcu_peripheral_check_usability/6`: la seguente regola Prolog, oltre ad utilizzare al suo interno la regola descritta in precedenza "`mcu_peripheral_check_avaiability/6`", si occupa di controllare che i PIN che i pin disponibili al collegamento siano congrui in base ai fatti che ne descrivono le loro caratteristiche; di fatto, oltre a verificare la disponibilità dei PIN, questa regola verifica anche la piena compatibilità funzionale tra questi, così che alla configurazione generata possa corrispondere all'atto pratico un nodo perfettamente operativo all'interno del sistema e del sistema di regole che ne fa uso.
- `mcu_get_conf/2`: data una particolare MCU, la seguente regola restituisce una struttura dati costituita da una lista di liste (nel seguente formato [*"McuName", "McuPin", "McuPinFunction", "PeripheralName", "PeripheralPin", "PeripheralPinFunction"*]), capace di rappresentarne la configurazione.
- `mcu_get_peripheral/2`: data una particolare MCU, ed un particolare dispositivo, restituisce esito positivo se esiste una connessione tra loro, indipendentemente dalle caratteristiche della connessione stessa.
- `mcu_gen_conf_code/2`: a partire da una particolare MCU, questa regola serve a fornire una lista contenete al suo interno tutto il codice simbolico utile all'inizializzazione della MCU.

8.2 SYSTEM

All'interno del livello SYSTEM sono gestiti tutti gli aspetti relativi al Network ed alla Crittografia tramite l'uso di regole e fatti dedicati alla connessione, invio dei messaggi attraverso un canale sicuro capace di garantire segretezza grazie all'ausilio della cifratura delle istruzioni inviate ed alla decifratura delle risposte ricevute da parte dei vari nodi.

8.2.1 network.pl

In questo file sono presenti tra i sorgenti delle regole che fanno uso del modulo "*library(socket)*" già presente in SWI-Prolog, ed in quanto tale ne è richiesta l'importazione. Le regole definite offrono un'ampia serie di funzionalità utili all'invio di stringhe simboliche (cifrate o in chiaro, a seconda dell'impostazione selezionata), la creazione di un client TCP o UDP, e la corretta ricezione delle risposte, qualora sia richiesto:

- `net_connect/4`: regola Prolog utile alla creazione di una connessione – di fatto uno Stream, restituito in risposta dalla stessa regola – specificandone come parametri, l'indirizzo dell'Host (il nodo), una porta ed una Socket; richiede l'importazione del modulo socket già presente all'interno delle librerie SWI-Prolog.
- `net_disconnect/1`: la seguente regola si limita ad interrompere una connessione esistente chiudendo uno Stream aperto e che la funzione richiede gli venga passato per parametro.
- `net_send/2`: la seguente regola, facendo uso della regola "`net_send/3`" descritta in seguito, risulta utile all'invio di un messaggio unicamente in chiaro, tramite il settaggio di uno dei parametri richiesti ("*Key*") al suo valore di default ("none").
- `net_send/3`: questa regola si occupa di inviare un dato messaggio seguiti dai due caratteri CR LF – rispettivamente carrigae return e line feed (0x0D0A) – attraverso l'uso di uno Stream e del parametro "*Key*" per la cifratura del messaggio. L'uso del valore "none" per il parametro "*Key*" richiesto dalla seguente regola comporta l'invio del messaggio direttamente in chiaro; per la seguente casistica esiste una regola dedicata allo scopo,

ovvero la "`net_send/2`" descritta poc'anzi. L'uso dei due caratteri non stampabili indicati sopra risulta utile all'interprete FORTH ospitato nella MCU in quanto utilizzati per la corretta e completa interpretazione delle stringhe di codice che deve gestire.

- `net_send_list/3`: la seguente regola Prolog si occupa dell'invio di intere liste di messaggi verso specifiche MCU – di cui ogni elemento è una singola istruzione FORTH – facendo uso della regola "`net_send/3`" descritta in precedenza. La MCU si occupa dell'esecuzione dell'intera sequenza di istruzioni nello stesso ordine con cui le sono state passate dal sistema di regole.
- `net_send_list/4`: a differenza della "`net_send_list/3`" descritta in precedenza, la seguente regola Prolog prevede la restituzione di una lista di pari lunghezza alla lista di comandi inviata, dove ad ogni suo elemento sarà eventualmente presente una stringa contenente la risposta corrispondente all'istruzione nella medesima posizione del messaggio generato ed inviato dalla logica artificiale.
- `net_receive/2`: regola utile alla ricezione dei messaggi unicamente in chiaro da parte della MCU facendo uso della regola "`net_receive/3`", descritta in seguito, settando uno dei parametri richiesti ("*Key*") al valore di default "none"; attraverso questa regola il sistema di regole interpreta direttamente il contenuto della risposta ricevuta senza prima procedere alla decodifica Base64 ed alla decifratura della stringa.
- `net_receive/3`: questa regola si occupa di gestire la ricezione di un dato messaggio di risposta seguiti dai due caratteri CR LF – rispettivamente carrigae return e line feed (0x0D0A) – attraverso l'uso di uno Stream e del parametro "*Key*" per la cifratura del messaggio. L'uso del valore "none" per il parametro "*Key*" richiesto dalla seguente regola comporta l'invio del messaggio direttamente in chiaro; per la seguente casistica esiste una regola dedicata allo scopo, ovvero la "`net_receive/2`" descritta poc'anzi. L'uso dei due caratteri non stampabili indicati sopra risulta utile all'interprete Prolog ospitato nell'unità centrale in quanto utilizzati come riferimento per la corretta e completa interpretazione delle stringhe di codice che deve gestire, ricevute dai nodi del sistema.

- `net_communication/4`: con la seguente regola si ha la possibilità di avviare una sessione di comunicazione con un server (in questo caso, il server REPL ospitato all'interno dalle varie MCU) così da poter inviare una lista di messaggi.
- `net_communication/5`: regola simile a quella descritta in precedenza, "`net_communication/4`", ma in più prevede la restituzione di una lista contenente le risposte ai messaggi inviati.

Entrambe le varianti della regola "`net_communication`", rispettivamente con arità 4 e 5, aprono a loro volta un thread parallelo che fa uso della regola "`net_communication_th`", rispettivamente nelle varianti con arità 3 e 4, così da poter garantire una comunicazione non bloccante tra il sistema di regole con i vari nodi del sistema; la differenza tra le due varianti della regola "`net_communication_th`", sta nel fatto che in quella con arità 4 è previsto un valore di ritorno restituito come risposta proveniente da una connessione a seguito dell'invio di un comando.

8.2.2 decenc.pl

Questo file contiene tutte le regole e i fatti specifici all'uso della cifratura e decifratura di stringhe da parte del sistema di regole, basandosi sugli algoritmi AES128-CBC o DES-CBC e l'algoritmo Base64 per la codifica/decodifica; tra i fatti principali di distinguono quelli relativi alle MasterKey (quindi attualmente memorizzate in chiaro), l'algoritmo di crittografia scelto, la dimensione del padding (dipendente dall'algoritmo di crittografia scelto ed il vettore di inizializzazione utile ai fini della crittografia CBC.

- `pad/2`: questa regola, con l'ausilio della regola "`pad_gen/3`", è utile ad includere per ogni messaggio, un padding di dimensioni variabili dato dalla dimensione del blocco meno il valore dato dal modulo della lunghezza del messaggio meno la dimensione del blocco.
- `gen_key/1`: regola utile alla generazione di una chiave random di dimensioni definite, da utilizzare come chiave di sessione.

- `prepare_digest/3`: questa regola si occupa di trattare il messaggio tramite codifica Base64, utilizzata dal sistema al fine di garantire una maggiore compatibilità con l'interprete FORTH nelle MCU.
- `enc/3`: la seguente regola effettua la cifratura di una stringa di simboli attraverso l'uso della Key impostata (MasterKey o SessionKey, se presente) e restituisce in uscita una lista dove ogni elemento è un blocco cifrato di dimensioni pari ad 8 o 16 caratteri (a seconda dell'algoritmo di cifratura scelto per la crittografia).
- `enc/4`: la seguente regola opera in modo analogo a quella descritta in precedenza, "`enc/3`", ma in più, dopo avere cifrato la sequenza simbolica, la processa tramite algoritmo di codifica in Base64.
- `dec/3`: la seguente regola effettua la decifratura di una stringa di simboli attraverso l'uso della Key impostata (MasterKey o SessionKey, se presente) e restituisce in uscita il testo chiaro generato come risultato dell'operazione svolta dalla MCU.
- `dec/4`: la seguente regola opera in modo analogo a quella descritta in precedenza, "`dec/3`" ma, prima di effettuare la decifratura, considera il messaggio ricevuto come codificato in Base64, in questo modo, prima di effettuare la decifratura del messaggio, lo processa tramite l'algoritmo di codifica/decodifica Base64.

8.3 APPLICATION

La descrizione di tutti gli oggetti presenti, di tutte le locazioni in cui si possono trovare gli oggetti o le periferiche così come i relativi percorsi per identificarli e raggiungerli, i parametri fisici dell'ambiente come pure lo stato che lo descrive, le azioni che si possono intraprendere sui vari oggetti presenti che sono in grado di influire parametri ambientali, fanno parte del livello APPLICATION e dei relativi file Prolog che la compongono.

8.3.1 object.pl

Gli oggetti rappresentano i dispositivi effettivamente presenti nel mondo e sono caratterizzati dalla loro tipologia (lamp, tv, ecc.), dalle classi di periferiche che possono controllarlo e/o monitorarlo e dalle azioni che è possibile eseguire attraverso essi; ricordiamo che uno degli obiettivi di questo lavoro è quello di simulare la possibilità di un interfacciamento diretto tra l'oggetto e l'utente senza che quest'ultimo necessiti di essere a conoscenza di tutta la parte hardware e software che sta nel mezzo.

- `object_type/1`: la seguente regola risulta utile alla definizione di una tipologia di oggetto che il sistema è in grado di gestire; alcune azioni possono essere specifiche per il dato oggetto, mentre altre possono essere comuni all'intera tipologia.
- `object_name/2`: la seguente regola risulta utile alla definizione di una istanza di oggetto specificandone il tipo ed il nome univoco che si desidera assegnare utile per la identificazione.
- `object_peripheral_class_usage/2`: tramite la seguente regola è possibile definire le classi di periferiche in grado di controllare una data tipologia di oggetto.
- `object_exec_function/3`: regola utile all'esecuzione di una determinata funzione su uno specifico oggetto; tra i parametri richiesti dalla regola si identifica la tipologia, il nome esatto (la regola si occupa di controllare anche se al nome corrisponde un oggetto della tipologia specificata) e l'azione da eseguire.

8.3.2 location.pl

Le locazioni ("*locations*") rappresentano luoghi del mondo reale in cui periferiche e oggetti si possono collocare ed attraverso il quale è poi possibile raggiungerle. Come già descritto nel capitolo 3 ("Sistema di regole e la base di conoscenza"), una location può contenerne al suo interno altre secondo una struttura annidata rappresentabile come albero, creando così un a serie di percorsi ("*path*") utili alla definizione di percorsi univoci per raggiungimento diretto di altre location, oggetti e di periferiche; un singolo percorso è rappresentato, all'interno della base di conoscenza, come una lista dove ogni elemento riesce a descrivere sempre più nello specifico cosa è possibile trovare al suo interno, fino al raggiungimento dell'ultimo elemento, ovvero un nodo foglia dell'albero contenente un oggetto, una periferica oppure una location (casistica di un percorso non completo).

- `location_name/1`: regola utile alla definizione di una istanza di location, specificandone dei nomi, non per forza univoci tra loro.
- `location_path/1`: regola utile alla definizione di un percorso all'interno della base di conoscenza.
- `location_add/2`: dato un percorso d'origine, la seguente regola aggiunge l'elemento desiderato all'interno di un percorso specificato effettuando l'assert di "`location_path/1`".
- `location_find_node/3`: dato un percorso (può essere dato in input anche senza specificarne tutti nodi esatti che lo compongono) e un elemento da trovare, restituisce una lista di percorsi validi in cui è possibile trovare quel dato elemento.
- `location_find_object/2`: dato un percorso in input da parte dell'utente, questa regola restituisce una lista contenente i percorsi completi per tutti gli oggetti presenti a partire dagli elementi del percorso specificati, considerandoli nell'ordine originale.
- `location_find_object/3`: regola analoga a quella già definita prima, "`location_find_object/2`", ma in più restituisce i percorsi degli oggetti che possono eseguire una determinata azione – passata per parametro – all'interno di quel percorso.
- `location_find_object/4`: il funzionamento di questa regola risulta simile a quello già descritto nelle regole "`location_find_object/2`"

è`location_find_object/3`", ma in più è consentito di specificare in fase di ricerca anche la tipologia di oggetto interessata.

- `location_find_sensor/2`: agisce come nel caso già descritto per la regola "`location_find_object/2`", ma invece degli oggetti questa regola ricerca e restituisce dei sensori.
- `location_find_sensor/3`: agisce come nel caso già descritto per la regola "`location_find_sensor/2`", ma in più restituisce unicamente quei particolari sensori che possono monitorare uno specifico parametro indicato dall'utente come parametro.
- `location_find_sensor/4`: questa regola opera secondo modalità simili a quelle già descritte dalle regole "`location_find_sensor/2`" e "`location_find_sensor/3`", ma in più effettua la ricerca anche in base alla tipologia di periferica.

8.3.3 parameters.pl

I parametri rappresentano ciò che è fisicamente misurabile e controllabile all'interno dell'ambiente e dagli oggetti considerati all'interno del caso di studio da parte delle periferiche presenti.

- `parameter_name/1`: tramite questa regola è possibile definire un parametro fisico dell'ambientale.
- `unit_measure/1`: tramite questa regola è possibile definire una unità di misura con cui quantificare un parametro fisico rilevato dalle varie periferiche presenti nell'ambiente.
- `parameter_measure/2`: tramite la seguente regola si va a definire qual è l'unità di misura con il quale l'agente intelligente misura un particolare parametro fisico.

8.3.4 states.pl

Uno stato ambientale può essere rappresentato dalla presenza di una o più condizioni ambientali.

- `status_name/1`: attraverso la seguente regola è possibile definire un particolare stato ambientale specificandone il nome, di fatto richiesto come parametro da parte dell'utente.
- `status_parameter/5`: per uno Stato Ambientale, questa regola definisce per il particolare parametro ambientale, il valore di soglia e l'operazione di confronto utili in fase di confronto per verificare se quel particolare stato ambientale si sia presente o meno; uno stato ambientale può essere caratterizzato da più parametri, ognuno con le sue caratteristiche in termini di valori di soglia e metodologia di confronto.
- `status_parameters/2`: regola che ha il compito di restituire la lista dei parametri che rappresentano un determinato stato specificato come parametro da parte dell'utente.
- `status_parameters_all/2`: regola che agisce in modo analogo a quella descritta sopra, "`status_parameters/2`", in quanto è in grado di restituirne le stesse informazioni ma per tutti gli stati esistenti nella base di conoscenza del sistema.
- `status_function/4`: questa regola esegue una funzione all'interno di un percorso, per un particolare stato restituendo, ove previsto, una lista dove ogni elemento è a sua volta una lista che contiene – per ogni parametro di cui lo stato è composto – i seguenti elementi:
    - Una lista dei percorsi validi per il sensore o attuatore in grado di svolgere l'azione desiderata per quello specifico parametro.
    - Il nome del parametro di cui si desidera ottenere le informazioni da parte dei nodi presenti.
    - Il valore misurato ottenuto a seguito della misurazione del parametro tramite sensore o come feedback a seguito di un'azione compiuta da un attuatore.

8.3.5 actions.pl

Questo file contiene tutte le regole che l'utente finale potrebbe utilizzare per l'esecuzione di query Prolog; queste regole operano in maniera trasparente attraverso gli altri livelli descritti in precedenza e sono utili all'esecuzione di funzioni ad alto livello.

- `exec/2`: dato uno specifico percorso – una lista di locazioni – questa regola esegue una l'azione desiderata dall'utente tramite il coinvolgimento di tutti gli oggetti e le periferiche presenti in quel percorso e capaci di portare a termine l'azione specificata.
- `exec/3`: opera in modo analogo alla regola già descritta "`exec/2`", ma in più consente di specificare una tipologia di oggetto su cui si desidera eseguire l'azione specificata.
- `execr/3`: opera in modo analogo alla regola già descritta "`exec/3`", ma in più prevede la restituzione, ove previsto, di una stringa contente il valore di ritorno generato a seguito dell'esecuzione dell'azione (particolarmente utile, ad esempio, nel caso di operazioni di rilevamento dello stato e prelievo delle informazioni svolte dai sensori).
- `execr/4`: opera in modo analogo alla regola già descritta "`execr/3`", ma in più consente l'esecuzione dell'azione specificata sugli stati ambientali definiti all'interno della base di conoscenza.

9 Test

Dopo aver sviluppato il modulo capace di generare del codice FORTH attraverso un processo di pianificazione basato sulla base di conoscenza relativa all'ambiente, l'hardware ed il problema da risolvere, si è provveduto all'installazione di un ambiente FORTH su una MCU di sviluppo e si sono svolti diversi test mirati alla verifica della corretta configurazione ed all'esecuzione di alcuni comandi, il tutto sulla base di quelle che erano le disponibilità di componenti da collegare al micro-controllore.

Nei test descritti a seguire è stata utilizzata una dev board NodeMCU AMICA basata su ESP8266-12E con i GPIO su piedinatura standard (2,54mm); maggiori informazioni relative alla scheda in esame sono presenti nel capitolo 5 ("Componenti secondarie") di questo elaborato.

La configurazione utilizzata nei test è costituita da due attuatori LED ed un sensore di luminosità analogico GL5516; quindi si inviano al sistema di regole tutte le informazioni relative ai collegamenti desiderati, specificando la MCU, le periferiche, eventualmente i pin coinvolti nella connessione, la tipologia e la direzione del canale di comunicazione. Il sistema, verificata la compatibilità e l'effettiva possibilità del collegamento, provvede ad arricchire la base di conoscenza e a generare il codice da inviare direttamente alla MCU specifica.

Come da codice presente in seguito, sono state specificate sia configurazioni complete che parziali (maggiori informazioni sono disponibili nel capitolo 3, paragrafo 2, del seguente elaborato); nello specifico si sono voluti collegare tra loro, due istanze di relay ("*relay1*" e "*relay2*") ad una specifica istanza di ESP ("*myesp*"), attraverso i PIN 1 dei relay – censiti nella base di conoscenza come dotati di funzionalità "*digital_input*" – ed i pin 12 e 14 della MCU – censiti nella base di conoscenza come dotati di funzionalità "*digital_output*" –, più una istanza di sensore di luminosità ("*lumsens*") senza specificarne i pin e le relative modalità di funzionamento; in tal caso, il sistema di regole, asserisce e restituisce la prima configurazione valida basata sulle porte ancora disponibili nella MCU ("*myesp*"); nello specifico caso, il sensore di luminosità vuole un collegamento di tipo analogico "*adc*" e nella ESP è presente una sola porta capace di avere tale modalità di funzionamento.

```
mcu_init([
     [myesp, 12, digital_output, relay1, 1, digital_input],
     [myesp, 14, digital_output, relay2, 1, digital_input],
     [myesp, _, _, lumsens, _, _]
]),
```

Nel caso non fosse possibile effettuare il collegamento, il sistema di regole restituirebbe "FALSE" come valore di ritorno restituito per quella regola.

Per questo test, è stata presa in considerazione una casa con alcune stanze dove all'interno di esse sono state posizionati oggetti e periferiche; tramite il codice presente di seguito, è stata definita all'interno della base di conoscenza una descrizione della suddivisione dell'ambiente di test, così come i percorsi utili per arrivare ai vari nodi dell'albero. Si noti che il primo elemento ("*home*") ha come primo elemento la lista vuota in quanto sopra di sé non è presente alcuna altra locazione, per questo tale elemento rappresenta la radice dell'albero, nonché la locazione più generale a cui ci si può riferire.

```
location_add([], home),
location_add([home], kitchen),
location_add([home, kitchen], table),
location_add([home, kitchen, table], lamp1),
location_add([home, kitchen, table], lumsens),
location_add([home], living),
location_add([home, living], table),
location_add([home, living, table], lamp2),
```

Ricordiamo che l'utente ha la possibilità di specificarli manualmente in fase di runtime, quindi senza la necessità di doverli inserire direttamente dentro il codice sorgente Prolog.

Gli oggetti definiti, e presenti nella struttura di locazioni come foglie, sono stati poi collegati agli attuatori in modo tale da dare la possibilità al sistema di regole di interagire con essi.

```
peripheral_init([
     [relay1, lamp1],
     [relay2, lamp2]
]),
```

Attraverso le informazioni asserite da tale procedura, il riferimento agli oggetti risulta essere ad alto livello: l'utente infatti non deve riferirsi alla MCU o alla periferica, ma direttamente all'oggetto consentendo una interazione più semplice. È il sistema di regole, tramite l'ausilio della base di conoscenza, a risalire alla combinazione MCU, collegamento e periferica ed a riuscire a interagire con quello specifico oggetto.

L'esempio di codice Prolog di seguito illustrato, rappresenta una query valida nel caso in cui si desidera gestire il parametro di luminosità (tramite i soli oggetti capaci di avere effetto su di esso, in questo caso delle lampade) a seconda di quello che è il parametro misurato in quel momento, confrontandolo opportunamente con un valore di soglia appartenente ad un possibile stato o ad una condizione obiettivo definita dall'utente.

```
execr([kitcken, lumsens], read, [Val1]),
atom_number(Val1, Num1),
Num1 > 200,
exec(kitchen, on)
```

A seguito di queste linee di codice Prolog, l'agente intelligente è risultato in grado di generare il codice FORTH e di inviarlo a tutti i dispositivi (periferiche e oggetti) appartenenti alla locazione specificata ("*kitcken*") capaci di modificare il parametro dello stato, il tutto dopo aver prelevato il valore da un sensore valido ("*lumsens*" all'interno di "*kitcken*") e confrontato con la soglia (>200) (previe le opportune operazioni di conversione del valore).

Ulteriore test è stato svolto tramite l'uso dalle seguenti regole Prolog:

```
exec(kitchen, off),
execr([kitchen, lumsens], read, [Val1]),
exec(kitchen, on),
execr([kitchen, lumsens], read, [Val2]),
exec(kitchen, off),
atom_number(Val1, Num1),
atom_number(Val2, Num2),
Num1 - Num2 > 500,
```

In questo caso, i valori registrati dal sensore di luminosità ("*lumsens*"), vengono prelevati con lo scopo di avere un feedback del corretto funzionamento, facendo un confronto dei valori registrati prima e dopo il tentativo di accensione della lampada (inizialmente spenta): i valori restituiti sono presenti nelle variabili "*Val1*" e "*Val2*" (terzo parametro della regola Prolog "*execr*") che, dopo le opportune conversioni, sono confrontati tra loro per misurare l'eventuale variazione da prima a dopo, ottenendo così l'informazione desiderata; nello specifico il test ha esito positivo se la differenza supera il valore di 500 (il sensore di luminosità analogico utilizzato durante le sperimentazioni è in grado di restituire valori variabili tra 0 e 1024 da intendere rispettivamente come valori di massima e minima luminosità ambientale).

Al fine di assicurare la sicurezza durante la trasmissione delle informazioni tra unità centrale ed MCU, è stato implementato un protocollo di comunicazione basato su due diversi algoritmi di cifratura, DES-CBC ed AES128-CBC; di fatto l'utente può scegliere quale dei due utilizzare in fase di produzione, tuttavia, anche per avere la possibilità di portare avanti dei test sperimentali e comparativi tra i due algoritmi sulla MCU ESP, sono state compilate ed alternate due immagini distinte della memoria flash, una con DES-CBC e l'altra con AES128-CBC.

Durante i test è stato preso in considerazione lo scenario in cui tutti gli elementi del sistema, unità centrale e le varie MCU usufruiscono del WI-FI (2,4Ghz) come canale di trasmissione, e che le componenti risiedono all'interno della stessa subnet di rete; in tale scenario si può fare affidamento ad un a rete già protetta da attacchi da parte di agenti esterni grazie all'ausilio del protocollo WPA2 che basa la sua elevata sicurezza sull'algoritmo di cifratura AES128-CBC; nonostante ciò si è scelto comunque di rendere sicura tale trasmissione, soprattutto visti anche i vari ambiti di utilizzo in cui tale progetto può essere destinato. Nel caso di applicazione in ambito aziendale, con una gran quantità di sistemi connessi ed utenti, e dove gli aspetti amministrati da parte del sistema possono andare ben oltre quelli amministrati nel caso di applicazione nel contesto della domotica (sicurezza, accessi in aree riservate, gestione degli impianti anti incendio o altro) ha senso privatizzare tali comunicazioni.

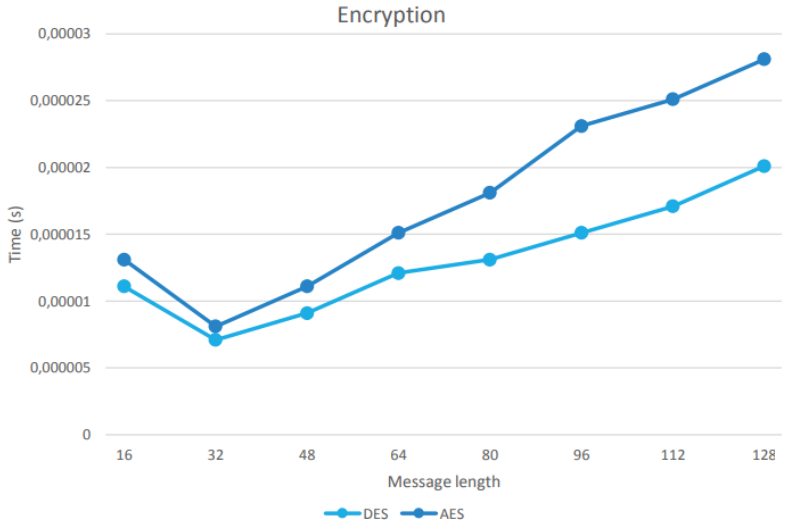
La scelta di usare delle chiavi simmetriche master per lo scambio di chiavi di sessione temporanee è stata presa in vista dei limiti (sia in termini di memoria che di performance) delle ESP8266. In prima analisi, si è cercato di implementare l'algoritmo DES-CBC direttamente in FORTH, ma il sistema è risultato instabile a causa dell'interprete utilizzato (nello specifico caso PunyForth); la limitata quantità di memoria disponibile dopo l'implementazione FORTH (causata soprattutto dal gran numero di matrici necessarie da DES per svolgere le varie permutazioni e rotazioni) rendeva impossibile l'esecuzione di istruzioni e dichiarazioni supplementari, causando il crash ed il reset delle MCU.

A seguito di tale situazione, si è scelto di implementare l'algoritmo direttamente in C, quindi di compilare l'immagine delle MCU con al suo interno i due algoritmi, e di inserire delle nuove istruzioni FORTH dedicate direttamente nel dizionario, "`>des`", "`des>`", "`>aes`", "`aes>`" rispettivamente per la cifratura e la decifratura svolta con i due algoritmi.

A primo impatto non si notano delle differenze sostanziali nella dimensione dell'immagine generata dal processo di compilazione, pari a 312kb sia per quella con l'implementazione di AES128-CBC che per quella con l'implementazione di DES-CBC (si tenga presente che entrambe le immagini sono da considerare comprensive dell'implementazione dell'algoritmo di codifica Base64). Le dimensioni di partenza dell'immagine, senza l'implementazione di qualsiasi algoritmo di cifratura e codifica, risulta essere pari a 304kb.

Dal punto di vista delle prestazioni, DES-CBC risulta più veloce rispetto ad AES128-CBC che però ricordiamo risulta essere meno sicuro di AES (nonostante sia la variante a 128bit implementata); relativamente al seguente aspetto sono stati svolti dei test comparativi eseguiti su normali personal computer (Intel Core i7-5557U (2 core), HT (4 thread), 3.10GHz, 256KB, cache L2, 4MB cache L3, con 16GB DDR3 (2GB for VM)). Sfortunatamente non è stato possibile svolgere i test direttamente nelle MCU in quanto nelle loro API non erano disponibili le istruzioni utili all'ottenimento dei tempi trascorsi (sarebbe stato possibile calcolare i tempi trascorsi da invio e ricezione dall'unità centrale, ma questi sarebbero stati influenzati dalle latenze dovute alle trasmissioni di rete).

Sono state svolte alcune operazioni di cifratura ed immediata successiva decifratura di stringhe di lunghezza variabile da 16 a 128 caratteri (limite del buffer del server REPL ospitato nella MCU).



	Encription	
Length	DES	AES
16	0,0000111	0,0000131
32	0,0000071	0,0000081
48	0,0000091	0,0000111
64	0,0000121	0,0000151
80	0,0000131	0,0000181
96	0,0000151	0,0000231
112	0,0000171	0,0000251
128	0,0000201	0,0000281

Fig. 16 Test performance DES e AES

I primi test svolti sulle stringhe di lunghezza 16 hanno sofferto dell'inizializzazione necessaria degli algoritmi, tuttavia in tutti i test svolti è possibile avere conferma delle migliori performance ottenibili da DES rispetto a quelle di AES che di contro, come già detto, offre maggiore sicurezza; differenze sostanziali sono tangibili già osservando la dimensione delle chiavi: 16 caratteri per AES contro i 8 caratteri per DES.

Come approccio all'implementazione, in entrambi gli algoritmi, la generazione delle chiavi è svolta separatamente dal processo di cifratura e decifratura; ad ogni comando di "*SET_KEY*" viene subito effettuata l'espansione delle chiavi che, una volta salvate in memoria, vengono riutilizzate ad ogni processo di cifratura/decifratura, fino a quando esse risulteranno valide. L'utente si trova quindi nelle condizioni di poter scegliere liberamente tra prestazioni e sicurezza a seconda delle sue necessità. Si tenga presente che il sistema utilizza delle chiavi di sessione; nonostante le vulnerabilità di DES, la validità/resistenza della chiave dovrebbe risultare superiore alla durata della chiave di sessione stessa (valida fino a prossimo rinnovo). Diverso è il caso in cui dovesse essere violata la MasterKey, l'unica accettata dalla MCU appena avviata fino ad un cambio tramite "*SET_KEY*".

Giusto appunto, le informazioni all'interno della memoria flash delle MCU non sono criptate, per questo un collegamento seriale al dispositivo permetterebbe di cercare la chiave poiché presente in chiaro all'interno della memoria flash.

Allo stesso tempo, un utente/amministratore con accesso fisico al dispositivo può modificare la MasterKey a suo piacimento con un periodo di validità che può essere temporaneo (fino al prossimo riavvio) o permanente (se viene creata una nuova immagine e caricata nella memoria flash dell'MCU).

10 Conclusioni e possibili sviluppi

Dai test svolti si è riusciti con successo a verificare la capacità del sistema di regole di gestire la corretta configurazione dei dispositivi embedded e di generare sequenze di codice simbolico funzionante, dando la possibilità anche all'utente di interagire con essi l'uso di una sintassi comune per qualsiasi MCU, il quale unico requisito è quello di avere un interprete FORTH al loro interno. Per di più si riesce ad interagire direttamente con gli oggetti del mondo in modo trasparente, senza che l'utente debba conoscere necessariamente la struttura specifica della MCU, collegamento e periferica coinvolta, in quanto basta specificare alla logica di intelligenza la sola configurazione desiderata (descritta anche ad alto livello) e la condizione obiettivo che si desidera ottenere.

Il sistema di regole è inoltre capace di operare in completa autonomia una volta che vengono definite le condizioni ambientali desiderate facendo ampio uso di una base di conoscenza in grado di discriminare e generalizzare tra le varie locazioni, oggettistica o, sempre più nello specifico, a livello di scheda (MCU, PIN dell'MCU e periferiche).

Oltre a delle istruzioni predefinite atte al monitoraggio dello stato degli oggetti collegati alle MCU (nel caso delle lampade utilizzate in fase di sperimentazione è bastato un sensore di luminosità, altri oggetti potrebbero necessitare di procedure più complesse e l'analisi contemporanea di più parametri) l'utente può sempre espandere la base di conoscenza in modo che la logica artificiale possa generare il codice da inviare direttamente alla MCU, idoneo al raggiungimento dell'obiettivo.

Il protocollo di comunicazione tra la logica artificiale e le varie MCU gode della presenza di due algoritmi di cifratura, DES-CBC e AES128-CBC, con chiavi di sessione (a pro della sicurezza generale del sistema) e della codifica Base64, liberamente disattivabili in caso di necessità; nulla impedisce l'uso di algoritmi diversi, sia di cifratura che di codifica, tuttavia in tal caso è necessario sia specificarli nella logica artificiale che implementarli direttamente nelle varie MCU (attualmente non è presente una regola Prolog che favorisca il cambio dell'algoritmo di crittografia da usare).

Attualmente le comunicazioni tra l'unità centrale e i vari nodi dislocati del nostro sistema avviene sfruttando il protocollo TCP ed, in caso di comunicazione contemporanea tra più nodi, non è attualmente previsto l'invio di comandi ed informazioni in broadcast, ciò nonostante è sempre possibile prevedere come eventuale sviluppo futuro, l'implementazione di tale funzionalità coadiuvato dal protocollo UDP; ciò richiede che l'utente tenga a mente l'attuale stato dell'arte del progetto, specie per quanto riguarda il protocollo di comunicazione attualmente presente: nello specifico, il sistema gestisce una chiave di sessione univoca per singolo nodo ed un messaggio inviato in broadcast potrebbe essere interpretato correttamente solo dai nodi che fanno uso di quella specifica chiave, attualmente generata in modo random dalla logica artificiale.

Ulteriore attenzione da avere, nel caso di implementazione di meccanismi broadcast per l'invio dei messaggi, è da concedere al servizio di autenticazione; allo stato dell'arte del sistema il servizio di autenticazione è basato sull'unicità della chiave (quindi del suo possessore), per questo occorre da parte dello sviluppatore, proporre tale servizio attraverso l'implementazione di ulteriori meccanismi, quale l'inclusione di una componente "nonce" o semplicemente dei codici incrementali all'interno dei soli messaggi specifici per MCU.

Nello specifico caso servirebbe che l'agente intelligente generi una chiave di sessione comune per tutti i dispositivi e che eventuali cambi della chiave valgano per tutti allo stesso modo, tuttavia è bene prestare attenzione all'eventuale overhead di informazioni inviati ad ogni rinnovo della chiave; l'aggiunta di una nuova componente periferica richiederebbe l'aggiornamento della chiave per tutti i dispositivi presenti e non solo per la componente aggiunta.

Appendice: Codice sorgente

In questa fase verranno elencati i sorgenti costituenti il progetto relativi alla realizzazione del sistema di regole ed i relativi contenuti in termini di fatti e regole che fanno parte della base di conoscenza; i seguenti sorgenti sono stati testati tramite l'interprete SWI-Prolog.

10.1 BOARD

Di seguito i sorgenti dei file Prolog dedicati alla descrizione a livello di BOARD dei PIN disponibili e delle relative funzionalità per ogni sensore, attuatore e MCU usati durante i test applicativi.

10.1.1 pin.pl

```
% Pin function types
pin_function_type(vcc).
pin_function_type(gnd).
pin_function_type(digital_input).
pin_function_type(digital_output).
pin_function_type(sda).
pin_function_type(scl).
pin_function_type(analog_input).
pin_function_type(analog_output).

% Pin function modes
pin_function_mode(digital_input, "GPIO_IN").
pin_function_mode(digital_output, "GPIO_OUT").
pin_function_mode(scl, "GPIO_OUT_OPEN_DRAIN").
```

```
% Pin function relations
pin_function_relation(digital_input, digital_output).
pin_function_relation(digital_output, digital_input).
pin_function_relation(analog_output, analog_input).
pin_function_relation(analog_input, analog_output).
pin_function_relation(scl, scl).
pin_function_relation(sda, sda).
pin_function_relation(vcc, vcc).
pin_function_relation(gnd, gnd).

pin_function_type_skip(vcc).
pin_function_type_skip(gnd).
```

10.1.2 peripheral.pl

```
:- dynamic peripheral_object_connection/2.
:- dynamic peripheral_name/2.

% Define peripheral
peripheral_class(relay).
peripheral_class(led).
peripheral_class(led_rgb).
peripheral_class(irda_tx).
peripheral_class(irda_rx).
peripheral_class(buzzer).
peripheral_class(dht).
peripheral_class(illuminance).

peripheral_type(relay_1).
peripheral_type(led).
peripheral_type(led_rgb).
peripheral_type(irda_tx).
peripheral_type(irda_rx).
peripheral_type(buzzer).
peripheral_type(dht11).
peripheral_type(dht22).
peripheral_type(gl5516).
```

```
peripheral_actuator(relay_1).
peripheral_actuator(led).
peripheral_actuator(led_rgb).
peripheral_actuator(irda_tx).
peripheral_actuator(buzzer).

peripheral_sensor(irda_rx).
peripheral_sensor(dht11).
peripheral_sensor(dht22).
peripheral_sensor(gl5516).

peripheral_actuator_parameter(led, illuminance).

peripheral_sensor_parameter(gl5516, illuminance).

peripheral_type_id(relay_1, 1).
peripheral_type_id(led, 2).
peripheral_type_id(led_rgb, 2).
peripheral_type_id(irda_tx, 3).
peripheral_type_id(irda_rx, 4).
peripheral_type_id(buzzer, 5).
peripheral_type_id(dht11, 6).
peripheral_type_id(dht22, 6).
peripheral_type_id(gl5516, 7).

peripheral_type_alias(relay_1, "RELAY").
peripheral_type_alias(led, "LED").
peripheral_type_alias(led_rgb, "LED").
peripheral_type_alias(irda_tx, "IRDATX").
peripheral_type_alias(irda_rx, "IRDARX").
peripheral_type_alias(buzzer, "BUZZER").

peripheral_type_class(relay_1, relay).
peripheral_type_class(led, led).
peripheral_type_class(led_rgb, led_rgb).
peripheral_type_class(irda_tx, irda_tx).
peripheral_type_class(irda_rx, irda_rx).
peripheral_type_class(active_buzzer, buzzer).
```

```
peripheral_type_class(passive_buzzer, buzzer).
peripheral_type_class(dht11, dht).
peripheral_type_class(dht22, dht).
peripheral_type_class(gl5516, illuminance).

peripheral_pin(relay_1, 1, digital_input, "S").
peripheral_pin(relay_1, 2, vcc, "VCC").
peripheral_pin(relay_1, 3, gnd, "GND").

peripheral_pin(led, 1, digital_input, "S").
peripheral_pin(led, 2, gnd, "GND").

peripheral_pin(led_rgb, 1, digital_input, "R").
peripheral_pin(led_rgb, 2, digital_input, "G").
peripheral_pin(led_rgb, 3, digital_input, "B").
peripheral_pin(led_rgb, 4, gnd, "GND").

peripheral_pin(irda_tx, 1, vcc, "VCC").
peripheral_pin(irda_tx, 2, gnd, "GND").
peripheral_pin(irda_tx, 3, digital_input, "S").

peripheral_pin(irda_rx, 1, vcc, "VCC").
peripheral_pin(irda_rx, 2, gnd, "GND").
peripheral_pin(irda_rx, 3, digital_output, "S").

peripheral_pin(active_buzzer, 1, digital_input, "S").
peripheral_pin(active_buzzer, 2, gnd, "GND").

peripheral_pin(passive_buzzer, 1, digital_input, "S").
peripheral_pin(passive_buzzer, 2, gnd, "GND").

peripheral_pin(dht11, 1, vcc, "VCC").
peripheral_pin(dht11, 2, gnd, "GND").
peripheral_pin(dht11, 3, scl, "S").

peripheral_pin(dht22, 1, vcc, "VCC").
peripheral_pin(dht22, 2, gnd, "GND").
peripheral_pin(dht22, 3, scl, "S").
```

```
peripheral_pin(gl5516, 1, vcc, "VCC").
peripheral_pin(gl5516, 2, gnd, "GND").
peripheral_pin(gl5516, 3, analog_output, "S").

peripheral_set_pin_mode_procedure([], []) :- !.

peripheral_pin_set_mode_procedure([[_, McuPin,
McuPinFunction, _, _, _] | T], [McuPinInitStr |
ResultCodeList]) :-
    (
        \+ pin_function_mode(McuPinFunction, _), ! ;

        pin_function_mode(McuPinFunction, McuPinMode),
        swritef(McuPinInitStr, "%w %w SET_MODE", [McuPin,
McuPinMode]),
        peripheral_set_pin_mode_procedure(T, ResultCodeList)
    )
.

peripheral_pin_function_definition(high, Pin, Res) :-
    swritef(Res, "%w HIGH", [Pin])
.

peripheral_pin_function_definition(low, Pin, Res) :-
    swritef(Res, "%w LOW", [Pin])
.

peripheral_pin_function_definition(analog_read, _, Res) :-
    Res = "ANALOG_READ .enc"
.

peripheral_function_prepare_code(CodeList, ResultCodeList) :-
    flatten(CodeList, ResultCodeListTmp),
    bagof(El, (member(El, ResultCodeListTmp), El \= ""),
ResultCodeList)
.

peripheral_pin_function(led, on, 1, digital_input, high).
peripheral_pin_function(led, off, 1, digital_input, low).
```

```
peripheral_pin_function(relay_1, on, 1, digital_input, high).
peripheral_pin_function(relay_1, off, 1, digital_input, low).

peripheral_pin_function(led_rgb, on, 1, digital_input, high).
peripheral_pin_function(led_rgb, on, 2, digital_input, high).
peripheral_pin_function(led_rgb, on, 3, digital_input, high).

peripheral_pin_function(led_rgb, off, 1, digital_input, low).
peripheral_pin_function(led_rgb, off, 2, digital_input, low).
peripheral_pin_function(led_rgb, off, 3, digital_input, low).

peripheral_pin_function(gl5516, read, 3, analog_output,
analog_read).

peripheral_exec_function_gen_code(PeripheralName, Function,
McuName, ResultCodeList) :-
    peripheral_name(PeripheralName, PeripheralType),
    mcu_get_peripheral(McuName, PeripheralName),
    findall([McuPin, McuPinFunction,
PeripheralPinFunctionDefinition], (
            mcu_peripheral_pin_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction),
            peripheral_pin_function(PeripheralType, Function,
PeripheralPin, PeripheralPinFunction,
PeripheralPinFunctionDefinition)
        ), List
    ),

    peripheral_exec_function_gen_code_procedure(List,
ResultCodeListTmp),
    peripheral_function_prepare_code(ResultCodeListTmp,
ResultCodeList)
.
```

```
peripheral_exec_function_gen_code_procedure([], _) :- !.

peripheral_exec_function_gen_code_procedure([[McuPin,
McuPinFunction, PeripheralPinFunctionDefinition] | T],
[[PinSetModeCode, PinGenCode] | ResCodeList]) :-
    peripheral_pin_set_mode_procedure([[_, McuPin,
McuPinFunction, _, _, _]], PinSetModeCode),

peripheral_pin_function_definition(PeripheralPinFunctionDefin
ition, McuPin, PinGenCode),

    peripheral_exec_function_gen_code_procedure(T,
ResCodeList)
.

peripheral_exec_function(PeripheralName, Function) :-
    peripheral_exec_function_gen_code(PeripheralName,
Function, McuName, ResultCodeList),
    mcu_send_message(McuName, ResultCodeList)
.

peripheral_exec_function(PeripheralName, Function,
ResponseList) :-
    peripheral_exec_function_gen_code(PeripheralName,
Function, McuName, ResultCodeList),
    mcu_send_message(McuName, ResultCodeList,
ResponseListTmp), (
        bagof(El, (member(El, ResponseListTmp), El \= ""),
ResponseList), ! ; true
    )
.

peripheral_object_check_usability(PeripheralType, ObjectType)
:-
    peripheral_type_class(PeripheralType, PeripheralClass),
    object_peripheral_class_usage(ObjectType,
PeripheralClass)
.
```

```
peripheral_object_check_usability(PeripheralName,
PeripheralType, ObjectName, ObjectType) :-
    peripheral_name(PeripheralName, PeripheralType),
    object_name(ObjectName, ObjectType),
    peripheral_object_check_usability(PeripheralType,
ObjectType)
.

peripheral_object_make_connection(PeripheralName, ObjectName)
:-
    peripheral_object_connection(PeripheralName, ObjectName),
! ;
    peripheral_object_check_usability(PeripheralName, _,
ObjectName, _),
    assert(peripheral_object_connection(PeripheralName,
ObjectName)), !
.

peripheral_init(ObjectConnectionList) :-
    peripheral_init_procedure(ObjectConnectionList), ! ;
    peripheral_init_rollback(ObjectConnectionList)
.

peripheral_init(PeripheralType, ObjectTypeList, ResList) :-
    peripheral_init_procedure(PeripheralType, ObjectTypeList,
ResList)
.

peripheral_init_procedure([]) :- !.

peripheral_init_procedure([[PeripheralName, ObjectName] | T])
:-
    peripheral_object_make_connection(PeripheralName,
ObjectName),
    peripheral_init_procedure(T)
.

peripheral_init_procedure(_, [], []) :- !.
```

```
peripheral_init_procedure(PeripheralType, [ObjectType | T],
[List | ResList]) :-
    findall([PeripheralType, ObjectType],
peripheral_object_check_usability(PeripheralType,
ObjectType), List),
    peripheral_init_procedure(PeripheralType, T, ResList)
.

peripheral_init_rollback(PeripheralConnectionList) :-

peripheral_init_rollback_procedure(PeripheralConnectionList)
.

peripheral_init_rollback_procedure([]) :- !.

peripheral_init_rollback_procedure([[PeripheralName,
ObjectName] | T]) :-
    retract(peripheral_object_connection(PeripheralName,
ObjectName)),
    peripheral_init_rollback_procedure(T)
.
```

10.1.3 pin.pl

```
:- dynamic mcu_peripheral_pin_connection/6.
:- dynamic mcu_net_address/3.
:- dynamic mcu_net_key/3.

% MicroControllerUnit types
mcu_type(pi).
mcu_type(esp8266_12e).

% MCU's FORTH main files
mcu_file(pi, "forth/piinit.fs").
mcu_file(esp8266_12e, "forth/esp8266_init.fs").
```

```
% MCU's pin
mcu_pin(pi, 9, digital_input).
mcu_pin(pi, 9, digital_output).
mcu_pin(pi, 17, digital_input).
mcu_pin(pi, 17, digital_output).
mcu_pin(pi, 27, digital_input).
mcu_pin(pi, 27, digital_output).

mcu_pin(esp8266_12e, 17, vcc).
mcu_pin(esp8266_12e, 18, gnd).
mcu_pin(esp8266_12e, adc, analog_input).
mcu_pin(esp8266_12e, 0, digital_input).
mcu_pin(esp8266_12e, 0, digital_output).
%mcu_pin(esp8266_12e, 1, digital_input).
%mcu_pin(esp8266_12e, 1, digital_output).
%mcu_pin(esp8266_12e, 2, digital_input).
%mcu_pin(esp8266_12e, 2, digital_output).
mcu_pin(esp8266_12e, 3, digital_input).
mcu_pin(esp8266_12e, 3, digital_output).
mcu_pin(esp8266_12e, 4, digital_input).
mcu_pin(esp8266_12e, 4, digital_output).
mcu_pin(esp8266_12e, 4, sda).
mcu_pin(esp8266_12e, 5, digital_input).
mcu_pin(esp8266_12e, 5, digital_output).
mcu_pin(esp8266_12e, 5, scl).
mcu_pin(esp8266_12e, 9, digital_input).
mcu_pin(esp8266_12e, 9, digital_output).
mcu_pin(esp8266_12e, 10, digital_input).
mcu_pin(esp8266_12e, 10, digital_output).
mcu_pin(esp8266_12e, 12, digital_input).
mcu_pin(esp8266_12e, 12, digital_output).
mcu_pin(esp8266_12e, 13, digital_input).
mcu_pin(esp8266_12e, 13, digital_output).
mcu_pin(esp8266_12e, 14, digital_input).
mcu_pin(esp8266_12e, 14, digital_output).
mcu_pin(esp8266_12e, 15, digital_input).
mcu_pin(esp8266_12e, 15, digital_output).
mcu_pin(esp8266_12e, 16, digital_input).
mcu_pin(esp8266_12e, 16, digital_output).
```

```
% Set MCU comunication params with network support (es,
ESP8266) with specified ip/port
mcu_net_address_assignment(McuName, Ip, Port) :-
    mcu_net_address(McuName, Ip, Port), ! ;
    assert(mcu_net_address(McuName, Ip, Port))
.

mcu_net_set_session_key(McuName, Message) :-
    gen_key(Key),
    strJoin(Key, " ", KeyMessage),
    strJoin([KeyMessage, "setkey"], " ", Message),
    mcu_send_message(McuName, [Message]),
    (
        mcu_net_key(McuName, _, session),
retract(mcu_net_key(McuName, _, session)), !;
        true
    ),
    string_chars(Key, KeyString),
    assert(mcu_net_key(McuName, KeyString, session))
.

% Get MCU Enc/Dec Key
mcu_net_get_key(McuName, Key) :-
    mcu_net_key(McuName, Key, session), ! ;
    mcu_net_key(McuName, Key, master), ! ;
    Key = none
.

% Send messages list to an MCU
mcu_send_message(McuName, MessagesList) :-
    mcu_net_address(McuName, Ip, Port),
    mcu_net_get_key(McuName, Key),
    net_communication(Ip, Port, Key, MessagesList)
.
```

```
% Send messages list to an MCU and store response in a list
mcu_send_message(McuName, MessagesList, ResponseList) :-
    mcu_net_address(McuName, Ip, Port),
    mcu_net_get_key(McuName, Key),
    net_communication(Ip, Port, Key, MessagesList,
ResponseList)
.

% Check if McuPin is free for attach a PeripheralPin
mcu_peripheral_check_usability(McuType, McuPin,
McuPinFunction, PeripheralType, PeripheralPin,
PeripheralPinFunction) :-
    mcu_pin(McuType, McuPin, McuPinFunction),
    peripheral_pin(PeripheralType, PeripheralPin,
PeripheralPinFunction, _),
    \+ pin_function_type_skip(McuPinFunction),
    \+ pin_function_type_skip(PeripheralPinFunction),
    pin_function_relation(McuPinFunction,
PeripheralPinFunction)
.

mcu_peripheral_check_avaiability(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction) :-
    mcu_name(McuName, McuType),
    peripheral_name(PeripheralName, PeripheralType),
    mcu_peripheral_check_usability(McuType, McuPin,
McuPinFunction, PeripheralType, PeripheralPin,
PeripheralPinFunction),
    \+ mcu_peripheral_pin_connection(McuName, McuPin, _, _,
_, _),
    \+ mcu_peripheral_pin_connection(_, _, _, PeripheralName,
PeripheralPin, _)
.
```

```
mcu_peripheral_make_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction) :-
    mcu_peripheral_pin_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction), ! ;
    mcu_peripheral_check_avaiability(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction),
    assert(mcu_peripheral_pin_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction))
.

mcu_init(PeripheralConnectionList) :-
    mcu_init_procedure(PeripheralConnectionList), ! ;
    mcu_init_rollback(PeripheralConnectionList)
.

mcu_init(McuType, PeripheralTypeList, ResList) :-
    mcu_init_procedure(McuType, PeripheralTypeList, ResList)
.

mcu_init_procedure([]) :- !.

mcu_init_procedure([[McuName, McuPin, McuPinFunction,
PeripheralName, PeripheralPin, PeripheralPinFunction] | T])
:-
    mcu_peripheral_make_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction),
    mcu_init_procedure(T)
.

mcu_init_procedure(_, [], []) :- !.
```

```
mcu_init_procedure(McuType, [PeripheralType | T], [List |
ResList]) :-
    findall([McuType, McuPin, McuPinFunction, PeripheralType,
PeripheralPin, PeripheralPinFunction],
mcu_peripheral_check_usability(McuType, McuPin,
McuPinFunction, PeripheralType, PeripheralPin,
PeripheralPinFunction), List),
    mcu_init_procedure(McuType, T, ResList)
.

mcu_init_rollback(PeripheralConnectionList) :-
    mcu_init_rollback_procedure(PeripheralConnectionList)
.

mcu_init_rollback_procedure([]) :- !.

mcu_init_rollback_procedure([[McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction] | T]) :-
    retract(mcu_peripheral_pin_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction)),
    mcu_init_rollback_procedure(T)
.


mcu_init_all(PeripheralConnectionList) :-
    mcu_init_all_procedure(PeripheralConnectionList,
PeripheralConnectionAllList),
    mcu_init_all_while(PeripheralConnectionAllList)
.

mcu_init_all_procedure([], []) :- !.
```

```
mcu_init_all_procedure([[McuName, McuPin, McuPinFunction,
PeripheralName, PeripheralPin, PeripheralPinFunction] | T],
[List | ResList]) :-
    findall([McuName, McuPin, McuPinFunction, PeripheralName,
PeripheralPin, PeripheralPinFunction],
mcu_peripheral_check_avaiability(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction), List),
    mcu_init_all_procedure(T, ResList)
.

mcu_init_all_while([]) :- !.

mcu_init_all_while([H | T]) :-
    mcu_init_procedure_while(H),
    mcu_init_all_while(T)
.

mcu_init_procedure_while([]) :- !.

mcu_init_procedure_while([[McuName, McuPin, McuPinFunction,
PeripheralName, PeripheralPin, PeripheralPinFunction] | T])
:-
    (mcu_peripheral_make_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction), ! ; true),
    mcu_init_procedure_while(T)
.

mcu_get_conf(McuName, ResList) :-
    findall([McuName, McuPin, McuPinFunction, PeripheralName,
PeripheralPin, PeripheralPinFunction],
mcu_peripheral_pin_connection(McuName, McuPin,
McuPinFunction, PeripheralName, PeripheralPin,
PeripheralPinFunction), ResList)
.
```

```
mcu_get_peripheral(McuName, PeripheralName) :-
    mcu_peripheral_pin_connection(McuName, _, _,
PeripheralName, _, _), !
.

mcu_gen_conf_code(McuName, McuCodeList) :-
    mcu_name(McuName, McuType),
    mcu_file(McuType, FILE_PATH),
    file_to_list(FILE_PATH, McuFileList),
    flatten([McuFileList], List),
    bagof(El, (member(El, List), El \= ""), McuCodeList)
.
```

10.2 SYSTEM

Di seguito i sorgenti dei file Prolog dedicati alla descrizione a livello SYSTEM dove sono gestiti tutti gli aspetti relativi al Network – connessione e comunicazione attraverso l'invio di messaggi – ed alla Crittografia – cifratura, decifratura e codifica delle istruzioni inviate e delle risposte ricevute verso e da parte dei vari nodi dislocati.

10.2.1 network.pl

```
:- use_module(library(socket)).

% Connection to specified host/port
net_connect(Host, Port, Socket, Stream) :-
    tcp_socket(Socket),
    tcp_connect(Socket, Host:Port),
    tcp_open_socket(Socket, Stream)
.

% Close connection
net_disconnect(Stream) :-
    close(Stream)
.

% Send messages through the network
net_send(Stream, Message) :-
    net_send(Stream, none, Message)
.

net_send(Stream, Key, Message) :-
    enc(Message, EncMessage, Key, digest),
    net_send_request_procedure(Stream, EncMessage)
.

net_send_request_procedure(_, []) :- !.

net_send_request_procedure(Stream, [Message | T]) :-
    writeln(Message), %FOR DEBUG
```

```
    string_concat(Message, "\r\n", Messageln),
    write(Stream, Messageln),
    flush_output(Stream),
    net_send_request_procedure(Stream, T)
.

% Send messages list through the network
net_send_list(_, _, []) :- !.

net_send_list(Stream, Key, [H | T]) :-
    net_send(Stream, Key, H),
    net_receive(Stream, Key, _),
%    writeln(Response), %FOR DEBUG
    net_send_list(Stream, Key, T)
.

net_send_list(_, _, [], []) :- !.

net_send_list(Stream, Key, [H | T], [Response |
ResponseList]) :-
    net_send(Stream, Key, H),
    net_receive(Stream, Key, Response),
%    writeln(Response), %FOR DEBUG
    net_send_list(Stream, Key, T, ResponseList)
.

% Recive messages from the network
net_receive(Stream, Res) :-
    net_receive(Stream, none, Res)
.

% Recive messages from the network
net_receive(Stream, Key, Res) :-
    read_line_to_codes(Stream, Output), (
        dec(Output, DecOutput, Key, digest), ! ;
        dec(Output, DecOutput, none, digest)
    ),
    string_chars(Res, DecOutput)
.
```

```
net_communication(Ip, Port, Key, MessagesList) :-
    net_connect(Ip, Port, _, Stream),
    net_communication_th(Stream, Key, MessagesList)
.

net_communication(Ip, Port, Key, MessagesList, ResponseList)
:-
    net_connect(Ip, Port, _, Stream),
    net_communication_th(Stream, Key, MessagesList,
ResponseList)
.

net_communication_th(Stream, Key, MessagesList) :-
    net_send_list(Stream, Key, MessagesList),

    net_send(Stream, "quit"),
    net_receive(Stream, _),
    net_disconnect(Stream)
.

net_communication_th(Stream, Key, MessagesList, ResponseList)
:-
    net_send_list(Stream, Key, MessagesList, ResponseList),

    net_send(Stream, "quit"),
    net_receive(Stream, _),
    net_disconnect(Stream)
.
```

10.2.2 decenc.pl

```
key('0123456789:;<=>?').

algorithm('aes-128-cbc').
iv([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]).
padvalue(0).

pad(Message, PadMessage) :-
    string_length(Message, MessageLength),
    Npad is 16 - mod(MessageLength, 16),
    pad(Message, PadMessage, Npad)
.

pad(Message, Message, 0) :- !.

pad(Message, PadMessage, Npad) :-
    pad_gen(PadList, Npad, 0),
    string_chars(Pad, PadList),
    strJoin([Message, Pad], "", PadMessage)
.

pad_gen([], Npad, Npad) :- !.

pad_gen([H | T], Npad, Counter) :-
    padvalue(H),
    NexCounter is Counter + 1,
    pad_gen(T, Npad, NexCounter)
.

enc(Message, EncMessage, none) :-
    EncMessage = Message, !
.

enc(Message, EncMessage, Key) :-
    pad(Message, PadMessage),
    algorithm(Algorithm),
    iv(IV),
```

```
    crypto_data_encrypt(PadMessage, Algorithm, Key, IV,
EncMessage, [padding(none)]), !
.

enc(Message, ResultList, none, digest) :-
    ResultList = [Message], !
.

enc(Message, ResultList, Key, digest) :-
    enc(Message, EncMessage, Key),
    prepare_digest(EncMessage, 16, ResultListTmp),
    append(ResultListTmp, ["\004\\004"], ResultList)
.

prepare_digest(Message, N, ResultList) :-
    sub_string_in_list(Message, N, MessageSliceList),
    prepare_digest_procedure(MessageSliceList, ResultList)
.

prepare_digest_procedure([], []) :- !.

prepare_digest_procedure([H | T], [MessageDigest |
ResultList]) :-
    base64(H, MessageDigest),
    prepare_digest_procedure(T, ResultList)
.

dec(EncMessage, Message, none) :-
    Message = EncMessage, !
.

dec(EncMessage, Message , Key) :-
    algorithm(Algorithm),
    iv(IV),
    crypto_data_decrypt(EncMessage, Algorithm, Key, IV,
DecMessage, [padding(none)]),
    replace_string(DecMessage, "\000", "", Message), !
.
```

```
dec(EncMessage, Message, none, digest) :-
    Message = EncMessage, !
.

dec(EncMessage, Message , Key, digest) :-
    string_codes(EncMessageString, EncMessage),
    base64(EncMessagePlain, EncMessageString),
    algorithm(Algorithm),
    iv(IV),
    crypto_data_decrypt(EncMessagePlain, Algorithm, Key, IV,
DecMessage, [padding(none)]),
    replace_string(DecMessage, "\000", "", Message), !
.

gen_key(Key) :-
    crypto_n_random_bytes(16, Key)
.
```

10.3 APPLICATION

Di seguito i sorgenti dei file Prolog dedicati alla descrizione del livello APPLICATION dove sono descritti tutti gli oggetti presenti, le locazioni in cui si possono trovare oggetti o periferiche così come i relativi percorsi per identificarli e raggiungerli, oltre i parametri fisici dell'ambiente in grado di descrivere uno stato, le azioni che si possono intraprendere sugli oggetti presenti in grado di influire i vari parametri ambientali.

10.3.1 object.pl

```
:- dynamic object_name/2.

% Classes of object managed
object_type(lamp).
object_type(led).
object_type(tv).

object_action(lamp, on).
object_action(lamp, off).
object_action(tv, send_signal).

object_peripheral_class_usage(lamp, relay).
object_peripheral_class_usage(lamp, led_rgb).
object_peripheral_class_usage(led, led).
object_peripheral_class_usage(tv, relay).
object_peripheral_class_usage(tv, irda_tx).

object_exec_function(lamp, ObjectName, on) :-
    object_name(ObjectName, lamp),
    object_action(lamp, on),

    peripheral_object_connection(PeripheralName, ObjectName),
    peripheral_name(PeripheralName, PeripheralType),
    peripheral_type_class(PeripheralType, PeripheralClass),
```

```
    object_peripheral_class_usage(lamp, PeripheralClass),

    peripheral_exec_function(PeripheralName, on,
ResponseList),
    print_list(ResponseList), !
.

object_exec_function(lamp, ObjectName, off) :-
    object_name(ObjectName, lamp),
    object_action(lamp, off),

    peripheral_object_connection(PeripheralName, ObjectName),
    peripheral_name(PeripheralName, PeripheralType),
    peripheral_type_class(PeripheralType, PeripheralClass),
    object_peripheral_class_usage(lamp, PeripheralClass),

    peripheral_exec_function(PeripheralName, off,
ResponseList),
    print_list(ResponseList), !
.
```

10.3.2 location.pl

```
:- dynamic location_name/1.
:- dynamic location_object/2.
:- dynamic location_path/1.

location_add(PathList, El) :-
    (location_name(El), ! ; object_name(El, _), ! ;
peripheral_name(El, _)),
     append(PathList, [El], ResList),
     (retract(location_path(ResList)), !; true),
     assert(location_path(ResList))
.
```

```
location_find_node(PathList, El, Res) :-
     is_list(PathList),
     findall(L, (location_path(L), intersection(PathList, L,
PathList), last(L, El)), Res),
     location_check_node_list(PathList, Res),
     !
.

location_find_node(LocationName, Leaf, Res) :-
     findall(L, (location_path(L), member(LocationName, L),
last(L, Leaf)), Res)
.

location_check_node_list(_, []).

location_check_node_list(PathList, [H | T]) :-
     list_check_order(PathList, PathList, H, -1, -1),
     location_check_node_list(PathList, T)
.

location_find_object(PathList, ResList) :-
    location_find_object(PathList, _, _, ResList)
.

location_find_object(PathList, Action, ResList) :-
    location_find_object(PathList, _, Action, ResList)
.

location_find_object(PathList, ObjectType, Action, ResList)
:-
    findall(Res, (
        object_name(ObjectName, ObjectType),
        object_action(ObjectType, Action),
        location_find_node(PathList, ObjectName, ResTmp),
        ResTmp \= [],
        flatten(ResTmp, Res)
    ), ResList)
.
```

```
location_find_sensor(PathList, ResList) :-
    location_find_sensor(PathList, _, _, ResList)
.

location_find_sensor(PathList, ParameterName, ResList) :-
    location_find_sensor(PathList, _, ParameterName, ResList)
.

location_find_sensor(PathList, PeripheralType, ParameterName,
ResList) :-
    findall(Res, (
        peripheral_name(PeriheralName, PeripheralType),
        peripheral_sensor_parameter(PeripheralType,
ParameterName),
        location_find_node(PathList, PeriheralName, ResTmp),
        ResTmp \= [],
        flatten(ResTmp, Res)
    ), ResList)
.
```

10.3.3 parameters.pl

```
% Managed parameters
parameter_name(temperature).
parameter_name(current).
parameter_name(umidity).
parameter_name(movement).
parameter_name(illuminance).
parameter_name(o).
parameter_name(co2).
parameter_name(pm10).
parameter_name(smoke).
parameter_name(flame).
parameter_name(infrared).
parameter_name(time).
```

```
% unit of measures
unit_measure(celsius).
unit_measure(relative_umidity).
unit_measure(ampere).
unit_measure(lux).
unit_measure(particle).
unit_measure(boolean).
unit_measure(seconds).

% Parameters unit of measure
parameter_measure(flame, boolean).
parameter_measure(temperature, celsius).
parameter_measure(temperature, kelvin).
parameter_measure(temperature, fahrenheit).
parameter_measure(umidity, relative_umidity).
parameter_measure(current, ampere).
parameter_measure(illuminance, lux).
parameter_measure(smoke, particle).
parameter_measure(time, seconds).
```

10.3.4 states.pl

```
% Define States: describes which parameters can define an
event in the real world
status_name(light).
status_name(dark).
status_name(fire).
status_name(current_flow).
status_name(smoke).

% States described by Parameters and Thresholdes
status_parameter(fire, temperature, 0, >, avg).
status_parameter(fire, flame, 0, >, none).
status_parameter(fire, smoke, 50, >, avg).

status_parameter(smoke, smoke, 50, >, none).
```

```
status_parameter(light, illuminance, 30, =<, avg).

status_parameter(dark, illuminance, 900, >, avg).

% Returns a List of Parameters of a given Status
status_parameters(Status, ParametersList) :-
    findall(Parameter, parameter(Status, Parameter, _, _, _),
ParametersList)
.

% Returns a complete infos List of Parameters of a given
Status
status_parameters_all(Status, ParametersList) :-
    findall(
        [Status, Parameter, Threshold, CompareOp,
StatisticOp],
        parameter(Status, Parameter, Threshold, CompareOp,
StatisticOp),
        ParametersList
    )
.

status_function(Path, StatusName, check, ResList) :-
    findall([SensorsResList, ParameterName, Threshold,
CompareOp, Op], (
        status_parameter(StatusName, ParameterName,
Threshold, CompareOp, Op),
        location_find_sensor(Path, ParameterName,
SensorsResList)
    ), SensorListToCall),

    status_function_procedure(SensorListToCall, check,
ResList)
.

status_function_procedure([], _, []) :- !.
```

```
status_function_procedure([[SensorsResList, ParameterName,
Threshold, CompareOp, _] | T], check, [ResListTmp | ResList])
:-
    status_function_sub_procedure(SensorsResList,
ParameterName, Threshold, CompareOp, ResListTmp),
    status_function_procedure(T, check, ResList)
.

status_function_sub_procedure([], _, _, _, []) :- !.

status_function_sub_procedure([SensorPath | T],
ParameterName, Threshold, CompareOp, [[SensorPath,
ParameterName, Delta] | ResList]) :-
    last(SensorPath, SensorName),
    execr(SensorName, read, [Val]),
    atom_number(Val, Num),
    deltaValues(Num, Threshold, Delta),
    status_function_sub_procedure(T, ParameterName,
Threshold, CompareOp, ResList)
.
```

10.3.5 actions.pl

```
% content to extend
action(on).
action(off).

exec(Path, ObjectType, Action) :-
    location_find_object(Path, ObjectType, Action, ResList),
    exec_procedure(ResList, Action), !
.

execr(Path, StatusName, Action, Response) :-
    status_name(StatusName),
```

```
    status_function(Path, StatusName, Action, Response), !
.

execr(PeripheralName, Action, Response) :-
    peripheral_name(PeripheralName, _),
    peripheral_exec_function(PeripheralName, Action,
Response), !
.

exec(Path, Action) :-
    location_find_object(Path, Action, ResList),
    exec_procedure(ResList, Action), !
.

exec_procedure([], _).

exec_procedure([H | T], Action) :-
    last(H, ObjectName),
    object_name(ObjectName, ObjectType),
    object_exec_function(ObjectType, ObjectName, Action),
    exec_procedure(T, Action)
.
```

10.4 Altri sorgenti

Di seguito sono presenti i sorgenti relativi ai file Prolog non direttamente classificabili tramite le categorie sopracitate; nello specifico si distinguono:

- *utils.pl*: sono presenti tutte le regole di uso generale utili al confronto, conversione, sostituzioni di stringhe, generazione di liste a partire dagli elementi presenti in un file di testo, verifica dell'ordine ed altre procedure dedicate alla visualizzazione ed alla manipolazione delle liste.
- *main.pl*: contiene un insieme di fatti e regole d'esempio utilizzate allo scopo di portare avanti dei test sperimentali di corretto funzionamento dei processi di deduzione da parte dell'agente intelligente .

10.4.1 utils.pl

```
removeDuplicates([], []).

removeDuplicates(List, Res) :-
    setof(X, member(X, List), Res).

cleanList(List, Res) :-
    flatten(List, ResTmp),
    removeDuplicates(ResTmp, Res)
.

deltaValues(V, S, R):-
    R is V - S
.

compareValuesBoolean(Val, O) :-
    O == >, Val > 0, ! ;
    O == <, Val < 0, ! ;
    O == >=, Val >= 0, ! ;
    O == <=, Val =< 0, ! ;
    O == ==, Val == 0, ! ;
    O == <>, Val \== 0, !
.
```

```
compareValues(Val1, Val2, Res) :-
    Val1 > Val2, Res = 1, !;
    Val1 == Val2, Res = 0, !;
    Val1 < Val2, Res = -1, !
.

compareListsLength(List1, List2, Res) :-
    length(List1, Length1),
    length(List2, Length2),
    compareValues(Length1, Length2, Res)
.

createPairsList(List1, List2, Res) :-
    compareListsLength(List1, List2, 0),
    createPair(List1, List2, Res)
.

createPair([], [], []).

createPair([H1 | T1], [H2 | T2], [ResTmp | Result]) :-
    ResTmp = [H1, H2],
    createPair( T1, T2, Result)
.

strJoin([], _, Empty) :-
    string_to_list(Empty, [])
.

strJoin([H|T], Separator, StrCat) :-
    strJoin(T, Separator, H, StrCat)
.

strJoin([], _, StrCat, StrCat).

strJoin([H|T], Sep, Str, Cat) :-
    string_concat(Sep, H, SepH),
    string_concat(Str, SepH, StrSepH),
    strJoin(T, Sep, StrSepH, Cat)
.
```

```
minList([H|T], Res) :-
    minList([H|T], H, Res)
.

minList([], ResTmp, ResTmp).

minList([H|T], ResTmp, Res) :-
    ResTmp2 is min(H, ResTmp),
    minList(T, ResTmp2, Res)
.

maxList([H|T], Res) :-
    maxList([H|T], H, Res)
.

maxList([], ResTmp, ResTmp).

maxList([H|T], ResTmp, Res) :-
    ResTmp2 is max(H, ResTmp),
    maxList(T, ResTmp2, Res)
.

avgList([H|T], Res) :-
    avgList([H|T], 0, 0, Res)
.

avgList([], Sum, Counter, Res) :-
    Res is Sum/Counter
.

avgList([H|T], Sum, Counter, Res) :-
    CounterTmp is Counter + 1,
    SumTmp is Sum + H,
    avgList(T, SumTmp, CounterTmp, Res)
.

print_list(List) :-
    print_list_procedure(List),
```

```
    format("~n")
.

print_list_procedure([]).

print_list_procedure([H | T]) :-
    print_list_procedure_current(H),
    print_list_procedure(T)
.

print_list_procedure_current(H) :-
    atom(H),
    format("~w ~n", H), !
.

print_list_procedure_current(H) :-
    string(H),
    format("~w ~n", H), !
.

print_list_procedure_current(H) :-
    number(H),
    format("~w ~n", H), !
.

print_list_procedure_current(H) :-
    print_list(H)
.

% Print file content
print_file_procedure(File) :-
    \+ at_end_of_stream(File),
    read_string(File, "\n", "\r", _, Line),
    format('~w~n', [Line]),
    print_file_procedure(File)
.
```

```
print_file(FILE_PATH) :-
    open(FILE_PATH, read, File),
    (print_file_procedure(File) ; true)
.

% Read file content and put on list
file_to_list_procedure(File, []) :-
    at_end_of_stream(File), !
.

file_to_list_procedure(File, [Line | List]) :-
    \+ at_end_of_stream(File),
    read_string(File, "\n", "\r", _, Line),
    file_to_list_procedure(File, List)
.

file_to_list(FILE_PATH, List) :-
    open(FILE_PATH, read, File),
    file_to_list_procedure(File, List)
.

replace_string(StringIn, Token, Replace, StringOut) :-
    split_string(StringIn, Token, "", List),
    strJoin(List, Replace, StringOut)
.

sub_string_in_list(String, N, ResultList) :-
    string_length(String, StringLength),
    sub_string_in_list_procedure(String, 0, N, StringLength,
ResultList)
.

sub_string_in_list_procedure(_, _, _, 0, []) :- !.
```

```
sub_string_in_list_procedure(String, I, N, _, [StringSlice |
ResultList]) :-
    sub_string(String, I, N, After, StringSlice),
    (After > N, Nnext is N, !; Nnext is After),
    Inext is I + N,
    sub_string_in_list_procedure(String, Inext, Nnext, After,
ResultList)
.

list_check_order([], _, _, _, _).

list_check_order([H | T], List1, List2, Ick1, Ick2) :-
     nth0(I1, List1, H),
     nth0(I2, List2, H),
     I1 > Ick1,
     I2 > Ick2,
     list_check_order(T, List1, List2, I1, I2)
.
```

10.4.2 main.pl

```
:- [utils, mcu, peripheral, network, object, forth, pin,
location, decenc, states, actions].

% Define MCUs, Peripherals and objects
mcu_name(myesp, esp8266_12e).
mcu_name(mypi, pi).

peripheral_name(rgb, led_rgb).
peripheral_name(relay1, relay_1).
peripheral_name(relay2, relay_1).
peripheral_name(lumsens, gl5516).

object_name(lamp1, lamp).
object_name(lamp2, lamp).

% Assign REPL Server IP:PORT
mcu_net_address(myesp, '192.168.1.XXX', 1983).
%mcu_net_address(myesp, 'host.name, 1983).
mcu_net_key(myesp, '0123456789:;<=>?', master).

% Define location
location_name(casa).
location_name(cucina).
location_name(salone).
location_name(tavolo).
location_name(altro).
location_name(sinistra).

test(ObjectName, R) :-
    (
        peripheral_object_connection(ActuatorName,
ObjectName),
        mcu_peripheral_pin_connection(McuName, _, _,
ActuatorName, _, _), ! ;
        mcu_peripheral_pin_connection(McuName, _, _,
ObjectName, _, _), ActuatorName = ObjectName
    ),
```

```
    peripheral_name(ActuatorName, Actuator),

    peripheral_actuator_parameter(Actuator, Parameter),
    peripheral_sensor_parameter(Sensor, Parameter),

    peripheral_name(SensorName, Sensor),
    mcu_peripheral_pin_connection(McuName, _,
McuSensorPinFunction, SensorName, _, _),

    location_object(A, B),peripheral_object_connection(P,
B),mcu_peripheral_pin_connection(M, _,_, P,_,_),

    upcase_atom(ActuatorName, ActuatorNameUC),
    McuSensorPinFunction = analog_input,
    strJoin([ActuatorNameUC, "OFF ANALOG_READ ."], " ",
Str1),
    strJoin([ActuatorNameUC, "ON 1000 MS ANALOG_READ ."], "
", Str2),
    strJoin([ActuatorNameUC, "OFF"], " ", Str3),
    mcu_send_message(McuName, [Str1, Str2, Str3], R)
.

switchif :-
    execr(lumsens, read, [Val1]),
    writeln(Val1),
    atom_number(Val1, Num1),
    Num1 > 200,
    exec(salone, on)
.
```

```
goal :-
    working_directory(_, "/pathToProject/src/"),

    location_add([], casa),
    location_add([casa], cucina),
    location_add([casa], salone),
    location_add([casa, cucina], tavolo),
    location_add([casa, salone], sinistra),
    location_add([casa, salone, sinistra], tavolo),
    location_add([casa, cucina, tavolo], altro),
    location_add([casa, cucina, tavolo, altro], lamp1),
    location_add([casa, cucina, tavolo, altro], lumsens),
    location_add([casa, salone, sinistra, tavolo], sinistra),
    location_add([casa, salone, sinistra, tavolo, sinistra],
lamp2),

    mcu_init([
        [myesp, 12, digital_output, relay1, 1,
digital_input],
        [myesp, 14, digital_output, relay2, 1,
digital_input],
        [myesp, adc, _, lumsens, 3, _]
    ]),

    peripheral_init([
        [relay1, lamp1],
        [relay2, lamp2]
    ]),

    mcu_gen_conf_code(myesp, MessagesList),
    print_list(MessagesList),
    mcu_send_message(myesp, MessagesList)
.
```

Riferimenti Bibliografici:

[1] A. Azzara, D. Alessandrelli, S. Bocchino, M. Petracca, P. Pagano, "PyoT a macroprogramming framework for the Internet of Things", *Proc. IEEE International Symposium on Industrial Embedded Systems*, pp. 96-103, Jun. 2014.

[2] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming wireless sensor networks using *Kairos*. In *Proceedings of the First IEEE international conference on Distributed Computing in Sensor Systems (DCOSS'05)*, Viktor K. Prasanna, Sitharama S. Iyengar, Paul G. Spirakis, and Matt Welsh (Eds.). Springer-Verlag, Berlin, Heidelberg, 126-140. DOI=http://dx.doi.org/10.1007/11502593_12

[3] Gaglio, S., Giuseppe Lo, R., Martorella, G., & Peri, D. (2017). DC4CD: "A Platform for Distributed Computing on Constrained Devices". ACM TRANSACTIONS ON EMBEDDED COMPUTING SYSTEMS, 17(1), 1-25.

[4] R. Calegari, E. Denti, S. Mariani, and A. Omicini, "Logic Programming as a Service (LPaaS): Intelligence for the IoT," in *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017)*, G. Fortino, M. Zhou, Z. Lukszo, A. V. Vasilakos, F. Basile, C. Palau, A. Liotta, M. P. Fanti, A. Guerrieri, and A. Vinci, Eds. IEEE, May 2017.

[5] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *OSDI, 2002*.

[6] A. Savvides, C. Han, and S. Srivastava, "Dynamic Fine-Grained localization in Ad-Hoc networks of sensors," MOBICOM, 2001.

[7] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," *SenSys, 2003*.

[8] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," *SenSys, 2004*.

[9] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring", *WSNA, 2002*.

[10] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, “Habitat monitoring: application driver for wireless communications technology,” *SIGCOMM Comput. Commun. Rev., 2001*.

[11] G. Kortuem, F. Kawsar, V. Sundramoorthy and D. Fitton, "Smart objects as building blocks for the Internet of things," in *IEEE Internet Computing*, vol. 14, no. 1, pp. 44-51, Jan.-Feb. 2010. doi: 10.1109/MIC.2009.143.

[12] A G. J. Nalepa and P. Ziecik, "Integrated Embedded Prolog Platform For Rule-based Control Systems," *Proceedings of the International Conference Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006.*, Gdynia, 2006, pp. 716-721. doi: 10.1109/MIXDES.2006.1706678.

[13] G. Aloi, G. Caliciuri, G. Fortino, R. Gravina, P. Pace, W. Russo, and C. Savaglio, "Enabling loT interoperability through opportunistic smartphone-based mobile gateways," *Journal of Network and Computer Applications*, vol. 81, pp. 74-84, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804516302405.

[14] Lin Li, Philipp Wagner, Ramesh Ramaswamy, Albrecht Mayer, Thomas Wild, and Andreas Herkersdorf. 2016. A Rule-based Methodology for Hardware Configuration Validation in Embedded Systems. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES '16)*, Sander Stuijk (Ed.). ACM, New York, NY, USA, 180-189. DOI=http://dx.doi.org/10.1145/2906363.2906377

[15] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. AAAI Press, 2013

[16] W. Ecker, W. Muller, and R. D ömer, editors. *Hardware-dependent Software*. Springer Netherlands, Dordrecht, 2009

[17] P. Wagner, L. Li, T. Wild, A. Mayer, and A. Herkersdorf. Knowledge-Based On-Chip Diagnosis for Multi-Core Systems-on-Chip. In *edaWorkshop 15*, pages 39–45, Dresden, Germany, May 2015.

[18] Christian Berger. 2014. SenseDSL: Automating the Integration of Sensors for MCU-Based Robots and Cyber-Physical Systems. In *Proceedings of the 14th Workshop on Domain-Specific Modeling (DSM '14)*. ACM, New York, NY, USA, 41-46. DOI=http://dx.doi.org/10.1145/2688447.2688455.

[19] Anton Ertl. 2011. Ways to reduce the stack depth. In *Proceedings of the 27th EuroForth Conference (EuroForth’11).* 36–41.

[20] Anton Ertl. 2013. PAF: A portable assembly language. In *Proceedings of the 29th EuroForth Conference (EuroForth'13)*. 30–38.

[21] Andrew Read. 2014. Concept and implementation of an extended return stack to enhance subroutine and exception handling in FORTH. In *Proceedings of the 30th EuroForth Conference (EuroForth'14)*. 5–22.

[22] Bill Stoddart, Campbell Ritchie, and Steve Dunne. 2012. Forth semantics for compiler verification. In *Proceedings of the 28th EuroForth Conference (EuroForth'12)*. 45–58

Indice delle figure