

Smart City – Project report

Enrico Benini, Nicola Casadei, and Marco Benedetti

Alma Mater Studiorum – University of Bologna

via Venezia 52, 47023 Cesena, Italy

{enrico.benini5, nicola.casadei4, marco.benedetti7}@studio.unibo.it

Table of Contents

Smart City – Project report	1
<i>Enrico Benini, Nicola Casadei, and Marco Benedetti</i>	
1 Introduzione	4
2 Visione	4
3 Obiettivi	4
4 Requisiti	5
4.1 Requisiti Non Funzionali	5
5 Acquisto Hardware	5
5.1 Dispositivi di Computazione	5
5.2 Sensori	7
5.3 Hardware Aggiuntivo	12
6 Analisi dei Requisiti	14
6.1 Casi D'Uso e Macro Operazioni	14
6.2 Scenario	15
6.2.1 Inserimento Range	15
6.2.2 Visualizzazione Stato Realtime	16
6.2.3 Visualizzazione Dati	16
6.3 Modello del Dominio	17
6.3.1 Sistema Embedded	18
6.3.2 Server	21
6.3.3 Web Site	26
7 Analisi del Problema	27
7.1 Architettura Logica	27

7.1.1 Sistema Embedded	28
7.1.2 Server	33
7.2 Gap di Astrazione	41
7.3 Analisi dei Rischi	41
8 Work Plan	42
8.1 Strumenti e Framework	43
9 Project	44
9.1 Database e Altre Rappresentazioni di Dati	44
9.2 Introduzione All'Architettura di Progetto	50
9.3 Struttura	51
9.4 Interazione	54
9.5 Comportamento	58
9.6 Struttura	59
9.7 Interazione	60
9.8 Comportamento	61
10 Configurazione Hardware	61
10.1 Schema di collegamento	61
11 Implementazione	63
12 Testing	63
12.1 Server	63

1 Introduzione

Questo è il report di progetto del corso di smart city dell'università di Bologna. Di seguito sarà consultabile tutto il processo di analisi del progetto: modelli, problemi riscontrati e soluzioni adottate, interazione con l'ambiente, sensori utilizzati e il loro collegamento ...

Per qualsiasi dubbio in merito fare riferimento agli autori.

2 Visione

La visione che guida questo progetto consiste nel raggiungere rapidamente l'ideale di città intelligente, quindi con un ambiente aumentato, capace di prendere decisioni e agire tempestivamente per far fronte a casi specifici e capace di comunicare direttamente con chi si trova immerso in esso, per facilitare la vita di tutti i giorni.

In particolare vogliamo prepararci ad imparare, modellare e costruire sistemi che si integreranno in questo contesto, visto l'andamento stesso del mercato che sta sempre più rendendo disponibili risorse di elaborazione e sensoristica a minor prezzo.

3 Obiettivi

Lo scopo del progetto è quello di implementare concretamente un'applicazione di domotica. Affrontando quindi tutte le problematiche ad essa annesse e fornire una possibile soluzione a queste. Ci auguriamo che questa possa essere di spunto per applicazioni simili e che possa quindi favorirne lo sviluppo.

Sfruttando questo progetto, vogliamo esplorare e apprendere la teoria e i concetti affrontati nel corso di smart city. Quindi tutti gli aspetti riguardanti la gestione di sensori e input provenienti dall'ambiente esterno. Uscendo dalla tipica zona di confort dei sistemi software centralizzati e non eterogenei.

4 Requisiti

Si vuole monitorare lo stato ambientale di una stanza. In particolare si vogliono monitorare lo stato di: luce, temperatura, umidità, gas e movimento, mantenendo la possibilità di aggiungere altre tipologie di sensori.

Il sistema dovrà dare all'utente la possibilità di inserire, attraverso un'interfaccia web, per ogni tipologia di dato, un'apposito range che indichi i valori ammessi all'interno della stanza in modo che, in caso uno dei valori misurati non risulti conforme alle specifiche, venga indicata una notifica di allarme sull'interfaccia stessa. Questo con l'idea di simulare la possibilità di eseguire delle azioni collegate all'allarme (ad esempio, accensione delle luci o del riscaldamento)

L'utente potrà inoltre visualizzare all'interno del sito i valori misurati in tempo reale e il valore dei vari sensori nel tempo.

4.1 Requisiti Non Funzionali

Aggiungiamo nei requisiti non funzionali tutte le proprietà che il sistema deve avere e che non sono state ufficialmente formalizzate.

- Separazione dei compiti e indipendenza tra le parti: garantire i principi SOLID dove è possibile evitare che un qualsiasi componente sia strettamente legato ad un'altro.
- Reattività: si visualizzi il Reactive Manifesto per i dettagli delle proprietà.

5 Acquisto Hardware

Sfortunatamente il primo problema incontrato in un progetto come il seguente è stata la necessità di acquistare la parte hardware del sistema che si andrà costruire. Di conseguenza si è messo in atto un processo di ricerca dei sensori, cavi e quant'altro per riuscire a soddisfare i requisiti

5.1 Dispositivi di Computazione

Prima di tutto necessitiamo di un dispositivo in grado di computare i dati emessi dai vari sensori e che sia interamente programmabile. Nel corso abbiamo visto due possibilità che hanno avuto molto successo recentemente:

- Arduino

- Raspberry Pi

Abbiamo scelto la seconda opzione data la maggior familiarità con il dispositivo e dal momento in cui risulta più facile il riutilizzo dello stesso una volta terminato questo progetto.

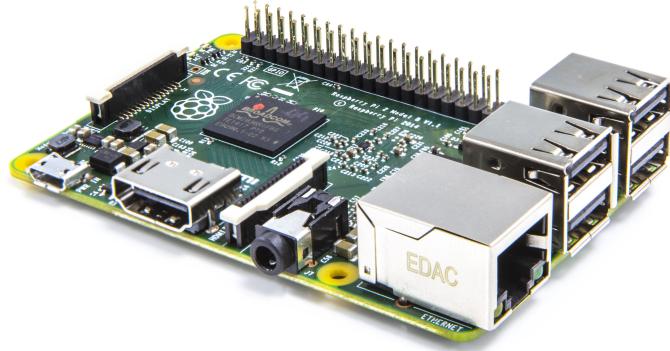


Fig. 1. Scheda del Raspberry 2

Il Raspberry Pi 2 (figura 1) è un SoC (sistema integrato) cioè possiede un chip che integra al suo interno processore, chipset, RAM ed eventuale circuiteria input/output. È dotato delle seguenti uscite:

- USB 2.0 (4)
- Ethernet
- HDMI
- Aux
- DC (Alimentazione)
- Slot MicroSD

Infine dispone di un GPIO (General Purpose Input/Output) interfaccia attraverso la quale è possibile comunicare con sensori esterni per mezzo di segnali digitali ed interagire con l'ambiente esterno.

Costo del dispositivo: 44,50 €

5.2 Sensori

Un'altra cosa fondamentale riguarda i sensori necessari per catturare i parametri richiesti. Tutti i sensori per standard possono lavorare con una tensione di 5v. Ogni sensore possiede 3 o 4 pin:

- Vcc: Pin di alimentazione
- GND: Pin della massa
- Dout: Porta input/output digitale
- Aout: Porta input/output analogica (opzionale)

Abbiamo Quindi scelto i sequenti sensori:

Parametri Ambientali	Sensori	Costo
Temperatura & Umidità	DHT11	6€
Luminosità	Light	5€
Movimento	HC-SR501	6€
Gas	MQ-2	7€

Table 1. Sensor table

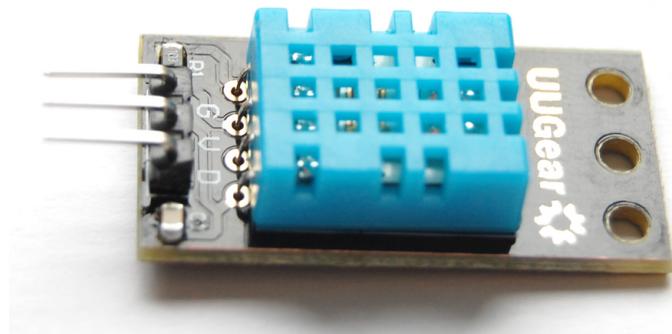


Fig. 2. DHT11 Sensor

Il sensore DHT11 (figura 2) è in grado rilevare la temperatura e l'umidità dell'ambiente circostante. Possiede tre Pin per interfacciarsi. Le sue caratteristiche tecniche sono:

- Vcc: 3.3~5.5V
- Range: Temperatura 0 ~50°C, Umidità: 20-90
- Accuratezza: Temperatura +-2°C, Umidità +-5
- Risoluzione: Temperatura 1°C, Umidità 1

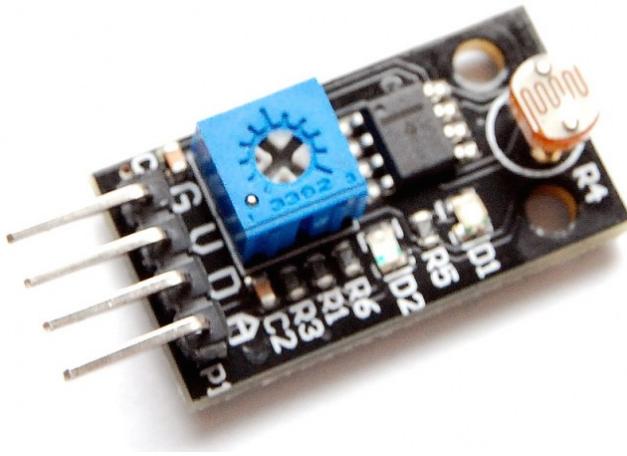


Fig. 3. Light Sensor

Il sensore di luminosità (figura 3) è un sensore dotato di un fotoresistore (componente circolare posizionato in fondo al circuito), un trimmer e due led. Il trimmer (equivalente ad un potenziometro) permette di regolare la sensibilità alla luce del sensore. Un led rimane costantemente acceso per indicare che il sensore è correttamente alimentato ed in funzione, l'altro si accende o si spegne nel momento in cui la luce percepita dal fotoresistore risulta sotto o sopra la soglia. Il sensore possiede quattro Pin per interfaciarsi. Le sue caratteristiche tecniche sono:

- Vcc: 3.3~5.5V
- Output: HIGH o LOW (boolean sensor: 0-1)



Fig. 4. HC-SR501 Sensor

Il sensore di prossimità (figura 4) permette di individuare gli spostamenti in una determinata area. Il suo funzionamento è dato da un componente chiamato "PIR" (Passive InfraRed) che per mezzo dei raggi infrarossi emanati dagli oggetti in uno spazio è in grado di percepire variazioni di movimento. Vi sono inoltre due trimmer arancioni per permettere di regolare il range di movimento e il tempo di delay. La semisfera di plastica trasparente posta sopra al modulo serve per poter massimizzare l'area di visibilità del sensore. Questa infatti, al suo interno, possiede particolari scanalature per che permettono di ridirigere correttamente i raggi al sensore. Vi sono tre Pin per interfacciarsi. Le sue caratteristiche tecniche sono:

- Vcc: 5~20V
- Range: 5-7 metri
- Ampiezza: 120°
- Tempo di delay : 0.3-600 secondi
- Output: HIGH o LOW (boolean sensor: 0-1)



Fig. 5. Gas sensor

Il sensore di gas (figura 5) è dotato di un sensore MQ-2 in grado di rilevare la presenza dei seguenti gas: LPG, propano, idrogeno. Possiede inoltre due led: uno sempre acceso per segnalare la giusta alimentazione del sensore e l'altro per segnalare la presenza di gas (acceso quando vi è gas). Infine, sotto, vi è un trimmer utile alla regolazione della sensibilità del sensore. Vi sono quattro Pin per interfacciarsi. Le sue caratteristiche tecniche sono:

- Vcc: 2,5~5V
- Output: HIGH o LOW (boolean sensor: 0-1)

5.3 Hardware Aggiuntivo

Abbiamo cercato di realizzare il progetto provando a semplificare il più possibile la parte hardware. In particolare ci siamo impegnati ad acquistare moduli (schede provviste di sensori e le resistenze necessarie al loro funzionamento) provvisti di sensore invece dei singoli sensori.

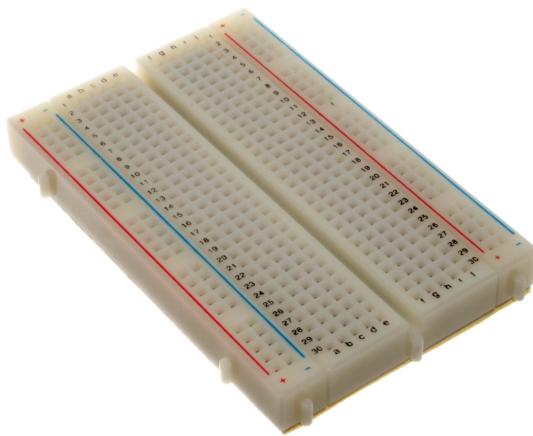


Fig. 6. Breadboard

Questo ci ha permesso di fare a meno di resistenze, led o quant'altro che rendesse necessario l'uso della breadboard (figura 6). Strumento in grado di effettuare collegamenti, utilizzato molto spesso per realizzare e testare prototipi di circuiti elettrici.

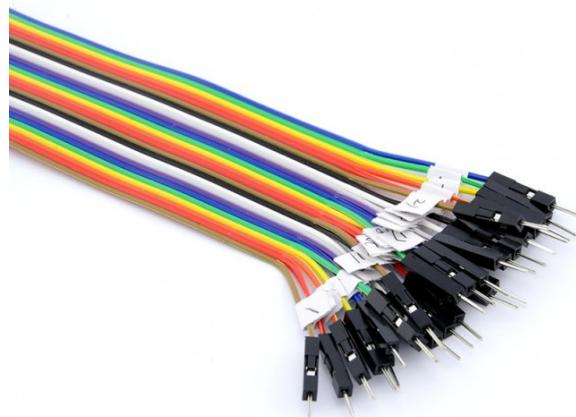


Fig. 7. Cavi di collegamento

Tuttavia è stato necessario acquistare un set di cavi di collegamento (esempio in figura 7), cioè cavi di metallo rivestiti in plastica, per permettere il collegamento dei vari pin dei sensori alla gpio del Raspberry.

Questi cavi possono essere di tre tipi a seconda delle combinazioni (maschio-maschio, maschio-femmina, femmina-femmina).

Il costo del set di cavi, composto da 50 cavi M-M , M-F, F-F è stato di 8€.

6 Analisi dei Requisiti

6.1 Casi D'Uso e Macro Operazioni

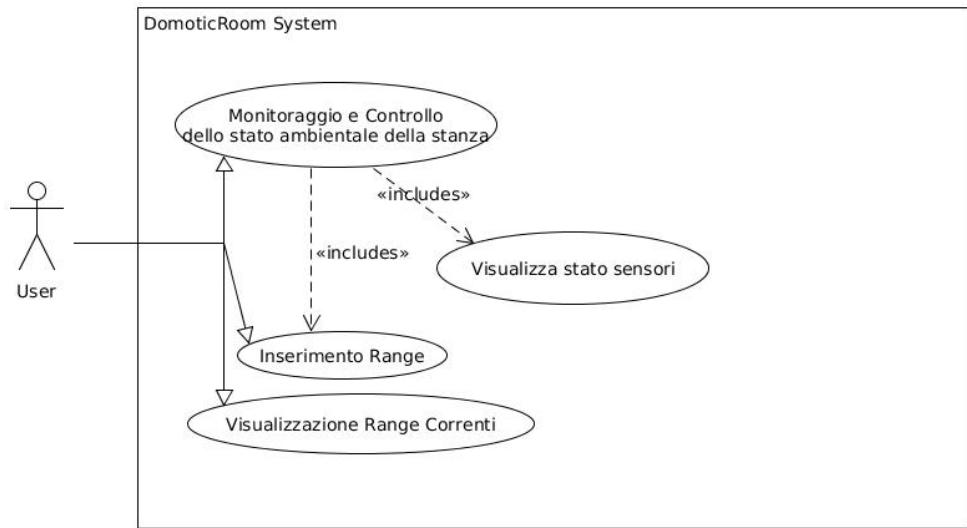


Fig. 8. Casi D'uso

Nell'immagine 8 si può osservare come l'utente interagisce con il sistema e quali sono gli obiettivi principali che si possono intuire dai requisiti. Questa è il punto di partenza per l'analisi di questi ultimi e quindi in quanto comunica immediatamente lo scopo del progetto.

Nell'immagine 9 si possono vedere le macro operazioni principali effettuate dal sistema e le interazioni con l'esterno. In particolare gli attori che interagiscono con il sistema saranno:

- La stanza: con questo attore si intendono i vari parametri che si possono rilevare attraverso i sensori e che quindi saranno di input per il sistema.
- Utente: con questo attore rappresenta l'utente che può interagire con il sistema.

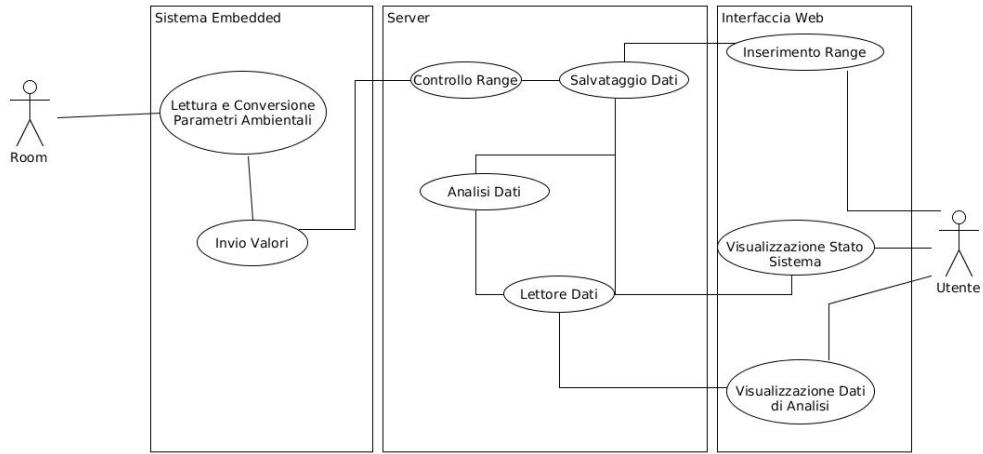


Fig. 9. Macro Operazioni

Il sistema è stato volutamente suddiviso in tre parti distinte con l'idea di seguire un modello MVC dove la parte di modello non viene aggiornata solamente attraverso l'input inserito dall'utente, ma anche e soprattutto dall'input dei sensori. L'organizzazione hardware ha fortemente influito su questa suddivisione.

È inoltre possibile visualizzare le macro operazioni effettuate e modellate dal sistema. Si vedano gli scenari di seguito per avere una più dettagliata visualizzazione dell'interazione tra le varie parti che lo schema sovrastante vuole rappresentare.

6.2 Scenario

In questa sezione verranno illustrati le principali modalità di utilizzo del sistema. Gli scenari elencati di seguito riguardano l'utente e di conseguenza si prevede l'accesso da parte di questo all'interfaccia di input.

Si prevede che il sistema sia opportunamente configurato e settato a livello hardware, senza errori durante la fase di start up.

6.2.1 Inserimento Range

1. Attraverso un apposito form l'utente è in grado di accedere alla funzionalità di settaggio dei range associati ai parametri ambientali.

2. Il server conosce già i sensori collegati al raspberry. Appena questi inviano qualche dato l'utente è in grado di visualizzare i controlli relativi ad ogni tipologia di sensore attualmente connesso. Conseguentemente l'utente è in grado di modificare tali intervalli.
3. Al termine della modifica degli intervalli l'utente dovrà confermare le modifiche attraverso un'apposito pulsante.
4. Il sistema mostra un messaggio di conferma o di errore.

6.2.2 Visualizzazione Stato Realtime

All'accesso del sistema l'utente visualizza lo stato realtime dei valori dei sensori ed eventuali notifiche:

- Se i valori vanno oltre gli intervalli correnti.
- Sullo stato dei sensori, se sono o meno attivi al momento.

In questa modalità l'utente non può effettuare alcuna operazione.

6.2.3 Visualizzazione Dati

Attraverso un apposito menù l'utente è in grado di accedere alla visualizzazione di alcuni semplici informazioni sui dati del sistema.

6.3 Modello del Dominio

In questa sezione vogliamo cercare di modellare le entità inserite all'interno dei requisiti senza fare riferimento alla parti tecnologiche e hardware, ma concentrando ci principalmente sui principali componenti del dominio. In questo modo siamo in grado di decidere le interfacce con cui vogliamo lavorare, costruendo la parte software, aumentando il disaccoppiamento con la parte fisica, aumentando anche la possibilità di utilizzare volendo lo stesso software su diverse configurazioni. In questo progetto, ad ogni modo, si utilizzerà solamente la configurazione descritta in questo report.

Suddivisione del Sistema: visto che è emerso già dai casi d'uso la separazione del sistema in varie parti, ci è sembrato giusto iniziare a modellare dividendolo fin da subito in modo da:

- semplificare il processo
- suddividere il lavoro di implementazione successivamente tra i membri del gruppo.

In particolare le parti individuate sono tre:

- Sistema Embedded: che nel nostro caso si occuperà di catturare, convertire e inviare i dati alle altre parti
- Server: Sarà la parte che riceve i dati e si occupa di effettuare le varie elaborazioni come ad esempio, il salvataggio dei dati, il calcolo delle statistiche e il controllo dei range. Infine questa parte dovrà rendere disponibili i risultati alla parte successiva.
- Web site: parte che, prendendo i risultati dalle parti precedenti, li mostra all'utente rispettando i casi d'uso predecenti.

Chiaramente in questa fase tutto viene semplificato e ridotto a poche entità. Questo però non significa che, ogni entità presente negli schemi sottostanti, non si riveli essere poi a sua volta un sottosistema più complesso. A sua volta in futuro puoàccadere di accorpate altre parti insieme. Di conseguenza si vuole mostrare il processo mentale effettuato in ogni fase.

Lo scopo di questa fase è appunto quella riflettere, in un primo modello, i requisiti. Successivamente, iniziare a sviluppare il progetto cercando di mantenere coerenza nelle interfacce principali che indicano le interazioni più importanti.

6.3.1 Sistema Embedded

Per quanto riguarda il sistema embedded è necessario prima effettuare delle indagini su come interagire e comunicare con i sensori in modo da modellare adeguatamente il tutto e quindi assicurarsi che in seguito sarà semplice riuscire a integrare il tutto con la tecnologia che avremo intenzione di utilizzare. Questa è una piccola eccezione che è necessario fare a questo livello. Tuttavia si fa riferimento a un paradigma più che ad una tecnologia specifica.

Data la nostra esperienza del corso e da vari esperimenti intendiamo l'interrogazione del **modello di interrogazione dei sensori è a polling** di conseguenza il nostro modello dovrà riflettere questa modalità di interazione. Sfortunatamente questo implica che a livello di modello è già **un'entità attiva** che si occupa di reperire i valori visto, che in una modalità a polling devo esplicitamente chiedere ai sensori i valori.

Struttura

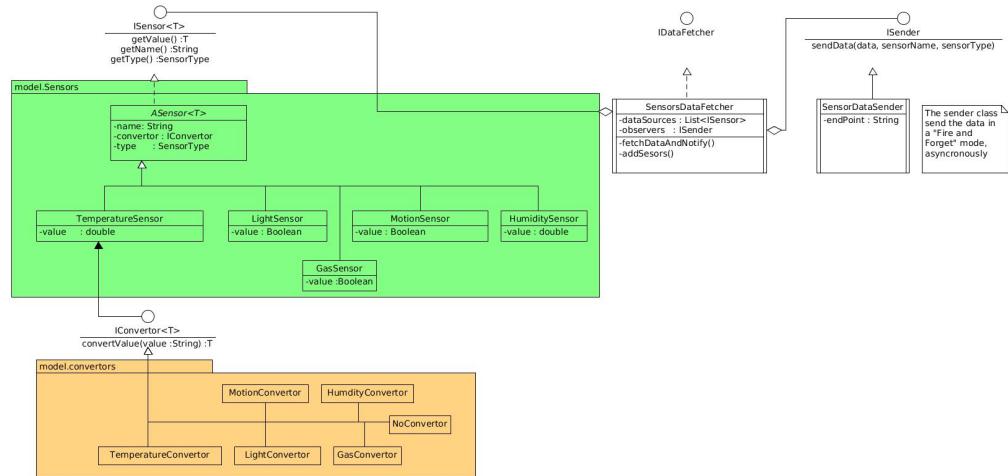


Fig. 10. Sistema Embedded, Struttura

Dalla struttura è possibile individuare subito le entità principali presenti nei requisiti. In particolare la presenza di una specifica gerarchia per i sensori

che condividono la stessa interfaccia che consente agilmente di ottenere il valore corrente del sensore. Sono stati inseriti solamente i sensori citati nei requisiti, ma si può facilmente intuire come qualsiasi tipo di sensore sia facilmente modellabile secondo questa struttura.

Si noti inoltre come viene anche inserita l'entità IConvertor che si occuperà di convertire il valore di uno specifico sensore in un'unità più consona per la sua gestione. Chiaramente questo viene affrontato fin da questo livello perché si immagina l'operazione di conversione come un'operazione quasi instantanea e necessaria.

Infine sono presenti le entità che si occupano, attivamente, di interrogare i sensori ogni intervallo di tempo predefinito e quindi di inviarli altrove. In questo caso è stato tutto ridotto ad un singolo endpoint, anche se poi possono essere facilmente più di uno. Come si può visionare nel commento, l'invio dei dati avviene con una metodologia di tipo *fire and forget*, quindi non avvengono reinvii dei dati e vengono ignorati eventuali errori. Chiaramente tutto questo è dovuto all'idea che i cicli di invio siano abbastanza brevi da potersi permettere eventuali perdite.

Interazione

Prima di iniziare si vuole evidenziare come viene riportato solamente un diagramma di interazione perché è presente solamente un'entità attiva. Tuttavia nel futuro potrebbero esserci più task e potranno essere necessari più diagrammi dell'interazione.

Nello schema di interazione vengono evidenziate le varie fasi del Sistema, in particolare la fase iniziale di setup, dove, conoscendo quali sensori sono presenti, questi vengono aggiunti nella memoria dell'entità principale in modo che in seguito siano facilmente interrogabili.

Terminata la fase di *Init* inizia il loop infinito che aspetta inizialmente un piccolo lasso di tempo per poi interrogare iterativamente tutti i sensori che sono stati aggiunti precedentemente, raccogliendo i dati e interrogando asincronamente l'entità di invio, per poi ricominciare il ciclo stesso.

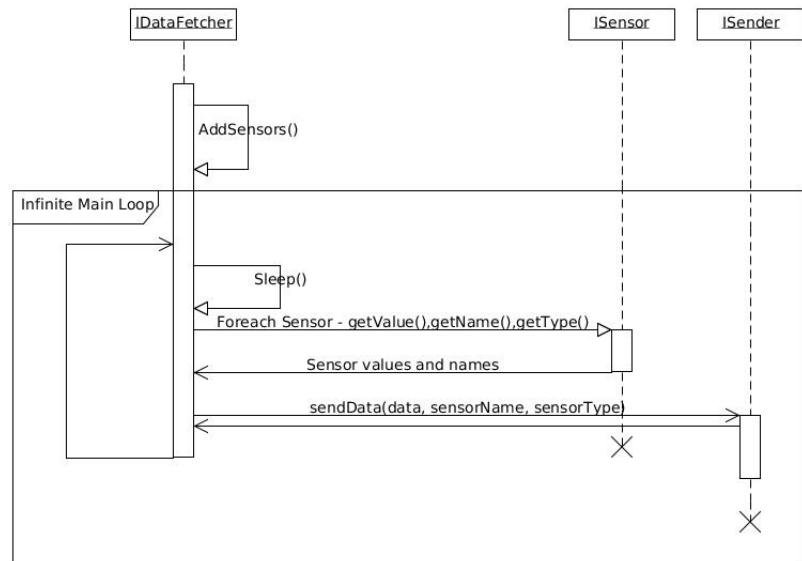


Fig. 11. Sistema Embedded, Interazione

Comportamento

Nel diagramma del comportamento viene semplicemente riportato quanto è stato precedentemente discusso attraverso una state machine.

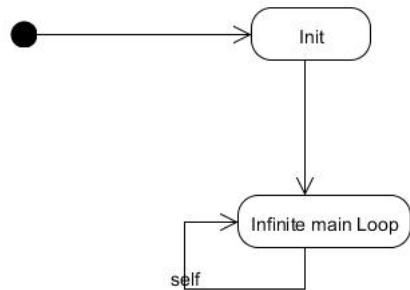


Fig. 12. Sistema Embedded, Comportamento

6.3.2 Server

La parte server è quella più importante di tutto il sistema in quanto è quella che concentra tutta la logica applicativa del sistema stesso.

Struttura

Le entità principali di questo schema sono:

- IPersistentStore, che si occuperà di salvare opportunamente i dati provenienti dai sensori e dall’utente
- IDataReceiver, che sarà sempre in ascolto per ogni messaggio proveniente dal sistema embedded e quindi notificherà opportunamente il sistema ad ogni nuovo arrivo
- IPresentator che si occuperà di ottenere i dati necessari per le viste da mostrare all’utente nel momento in cui una nuova richiesta verrà inoltrata.

Chiaramente ognuna di queste entità è in parte citata nei casi d’uso.

Si vedano i diagrammi dell’interazione per i dettagli di come avviene la comunicazione di tutte queste entità al fronte di garantire il funzionamento e il soddisfacimento dei requisiti.

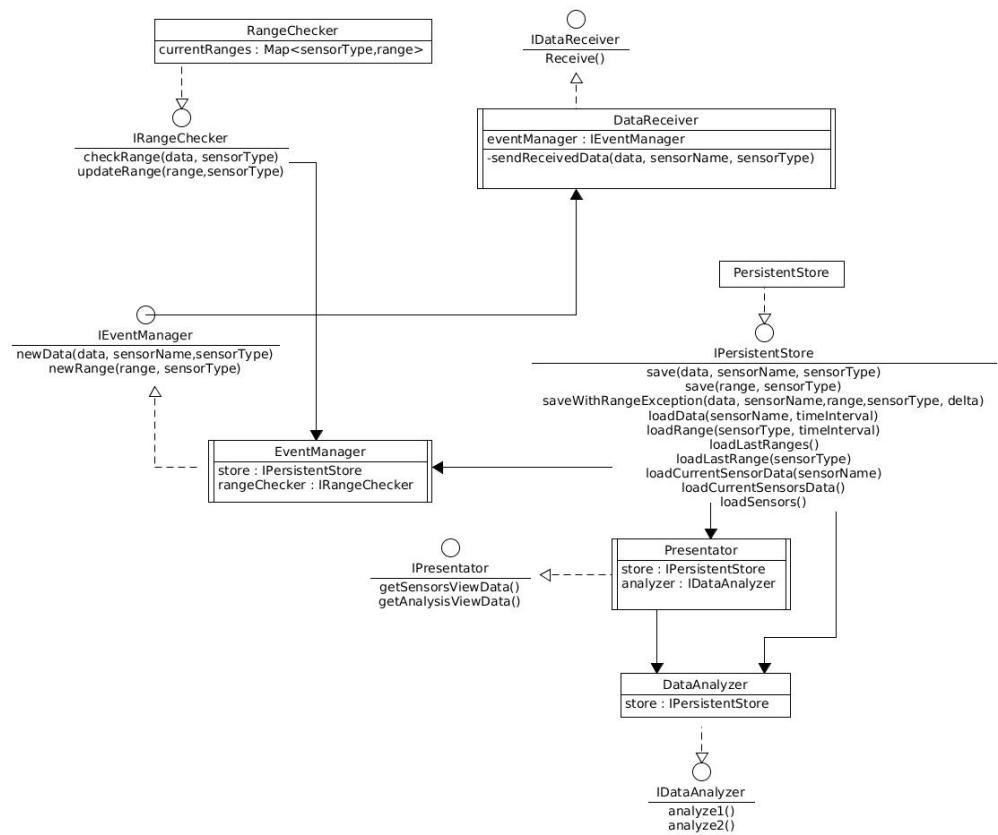


Fig. 13. Server, Struttura

Interazione

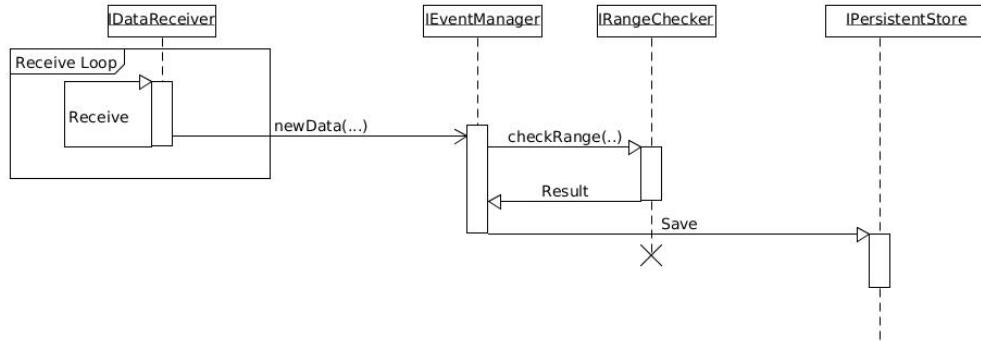


Fig. 14. Server, Interazione, Dati dai Sensori

Abbiamo deciso di omettere la parte di inizializzazione del sistema in questa fase dal momento che risulterà meglio definita nell'analisi del problema, dovendo aggiungere entità che non sono direttamente correlate con i requisiti principali ma con requisiti non funzionali.

Nello schema sovrastante si può osservare l'interazione nel caso in cui i sensori emettano un nuovo valore. Si vuole porre particolare enfasi su come l'entità che riceve i dati rimanga sempre in ascolto di nuovi arrivi delegando, tramite una chiamata asincrona alle altre entità, il compito di salvare i dati appena ricevuti.

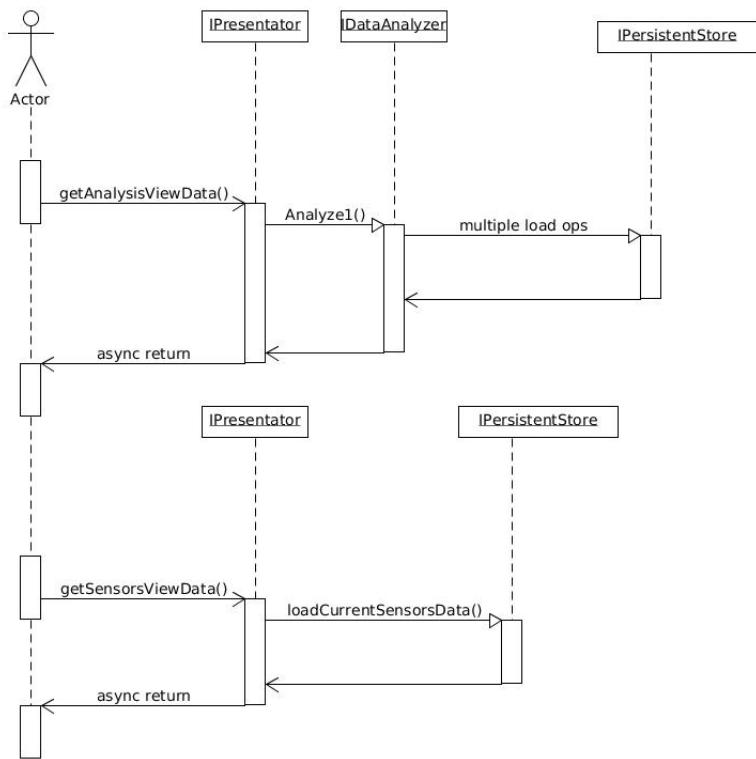


Fig. 15. Server, Interazione, Richiesta Visualizzazione Dati

In quest'altro schema dell'interazione si può osservare che entità vengono coinvolte a fronte di una chiamata di visualizzazione dati da parte dell'utente. Anche in questo caso la chiamata iniziale è stata effettuata in maniera asincrona in modo da lasciare il sistema libero di rispondere ad eventuali altre richieste per poterle gestire in maniera concorrente.

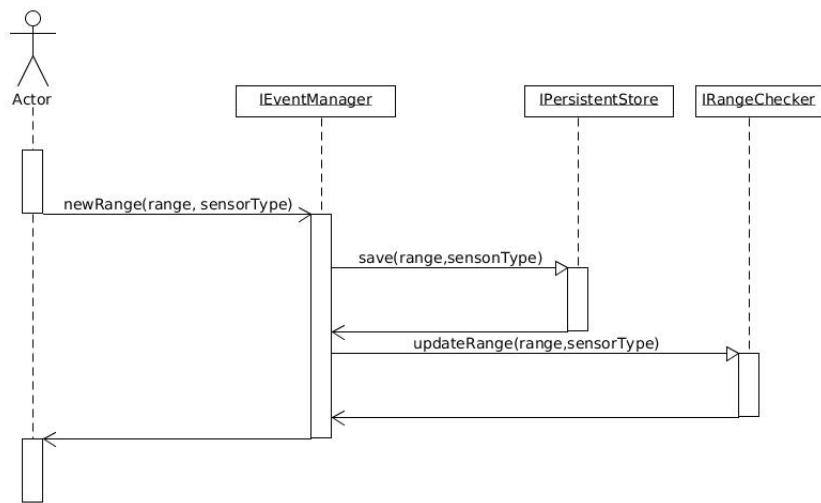


Fig. 16. Server, Interazione, Nuovo Range

In questo caso invece viene illustrata l'interazione per quanto riguarda la richiesta da parte dell'utente dell'inserimento di un nuovo range. Si nota come la chiamata, questa volta, arrivi all'entità che gestisce gli eventi essendo questo effettivamente un caso incui avviene un cambiamento nei dati e quindi tutto va gestito appropriatamente per evitare i conflitti.

Comportamento

In questa fase si mostrano le entità che sono effettivamente attive e che quindi operano cambiamenti sul sistema. Si è deciso di omettere tutte le entità che vengono solamente chiamate ed effettuano una unica operazione al fine di evitare di formalizzare troppe cose a questo livello.

Si vuole far notare inoltre come tutte queste entità abbiano effettivamente uno stato di attesa di un qualche messaggio o evento. Questo aspetto può essere molto importante in seguito nelle scelte per la progettazione.

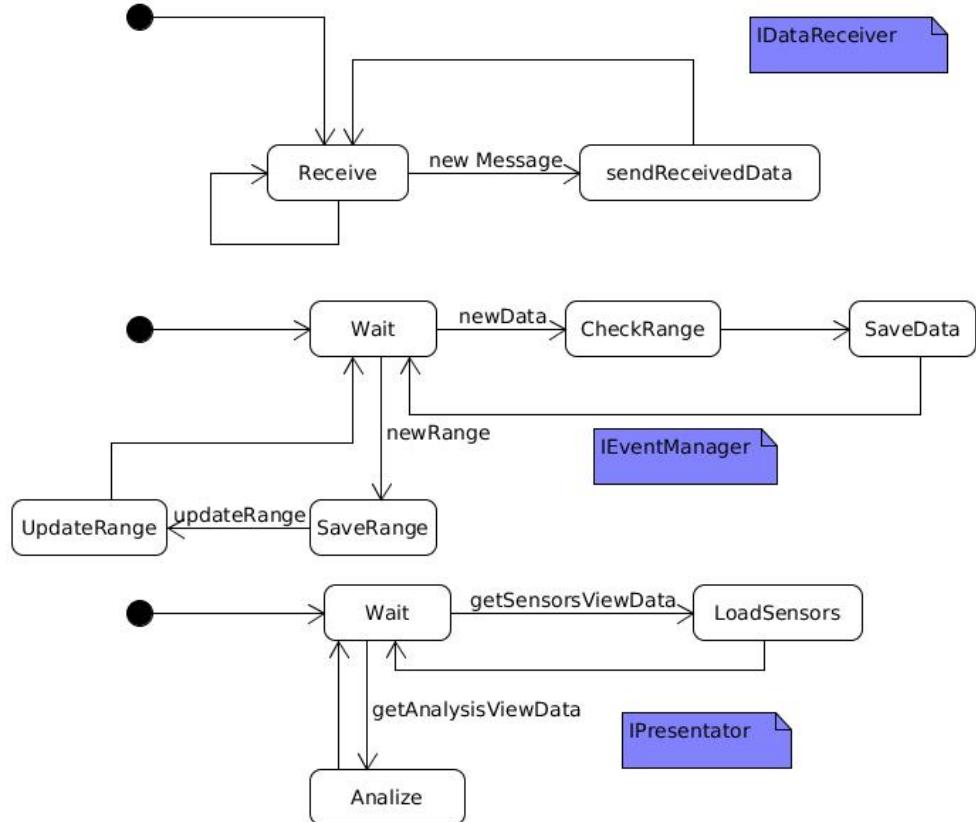


Fig. 17. Server, Comportamento

6.3.3 Web Site

Dopo un'attenta analisi abbiamo ritenuto superflua la necessità di modellare e analizzare la parte di visualizzazione in quanto non è effettivamente presente nessuna business logic ma solamente la parte che effettua le richieste di dati alle parti precedenti visualizzando tutto su schermo.

La documentazione relativa a questa parte si limiterà solamente a come utilizzare il software attraverso l'interfaccia.

Possiamo pensare di definire i link a cui l'interfaccia dovrà rispondere appropriatamente attraverso una pagina web. Questo ci consente di impostare

comunque dei test che esulano dal contenuto della pagina, garantendoci però che questa sia effettivamente online in maniera automatica.

gli URL scelti sono:

- *http://...../DomoticRoom/Status*
- *http://...../DomoticRoom/NewRange*
- *http://...../DomoticRoom/Analysis*

7 Analisi del Problema

Assunzioni Prima di iniziare l'analisi del problema abbiamo ritenuto necessario effettuare delle assuzioni riguardanti al paradigma che ci consentirebbero di effettuare un prodotto di qualità migliore e con meno sforzi.

Il paradigma di programmazione di riferimento è il *reactive programming* perché la concezione di flusso di dati è proprio quello che ci interessa modellare in quanto anche nel nostro sistema sarà presente un flusso di dati dal client al server. Per la comunicazione attraverso la rete questo ci consente di sfruttare chiamate asincrone aumentando il disaccoppiamento tra client e server.

Per questo abbiamo deciso di utilizzare per i dati un database NoSQL per via dell'estrema dinamicità, in quanto ci consente di aggiungere dei campi anche in seguito e un miglioramento di performance nell'accesso a dati che normalmente richiederebbero dei join.

Per ulteriori informazioni più dettagliate sul paradigma si rimanda ad un'altra sede.

7.1 Architettura Logica

Per la costruzione dell'architettura logica ci si baserà ovviamente sulla precedente analisi dei requisiti in modo da mantenere il contatto con questi e non rischiare di uscire fuori specifica. Anche in questo caso si andrà separare l'analisi in due parti:

- Sistema Embedded
- Server e Website

La parte del website è stata incorporata direttamente nella parte server proprio per la sua assenza di business logic.

Modellazione Reactive Prima di iniziare a costruire la modellazione sulla base del paradigma appena citato è necessario capire come fare a modellare lo stream stesso all'interno della nostra architettura logica, conseguentemente si è deciso di utilizzare i *marable diagrams*. Questi sono una rappresentazione di come il flusso di dati nel tempo avviene e quali tipologie di trasformazioni consentono di effettuare.

Con questi schemi ci viene concesso ancora una volta di concentrarci prima sul "cosa" e non sul "come" che invece viene delegata alla parte progettuale, dove si cercherà effettivamente di implementare il risultato finale dell'analisi

7.1.1 Sistema Embedded

Flussi Dati

Alla luce del paradigma di programmazione che si è deciso di adottare sono stati effettuati dei cambiamenti anche alla struttura che è stata esposta precedentemente proprio perché ci sono state fornite da questa assunzione nuovi livelli di astrazione, primo tra tutti il concetto di **Stream/Flusso**. Quindi in questa fase abbiamo ritenuto molto utile iniziare a visualizzare effettivamente questi flussi tramite un'apposito diagramma.

In particolare si puo' notare come in questo diagramma il compito di convertire i valori di un sensore sia stato disaccoppiato dal sensore stesso che quindi si deve solamente occupare di invitare i dati all'interno del flusso. È stato inoltre aggiunto anche una nuova entità *packager* che si occuperà di formattare i valori secondo il protocollo che deve essere definito tra client e server per l'apposita comunicazione. In particolare ogni dato proveniente da un sensore avrà un suo formato per via: della differenziazione dei dati, della tempistica di rilevazione che può anche risultare diversa e dell'idea per cui la comunicazione in ogni caso debba essere asincrona.

Sulla base di questa ultima frase si vuole sottolineare che ogni sensore quindi avrà un suo flusso asincrono, ma che in quest'immagine si è deciso di condensare in un'unica immagine per comodità.

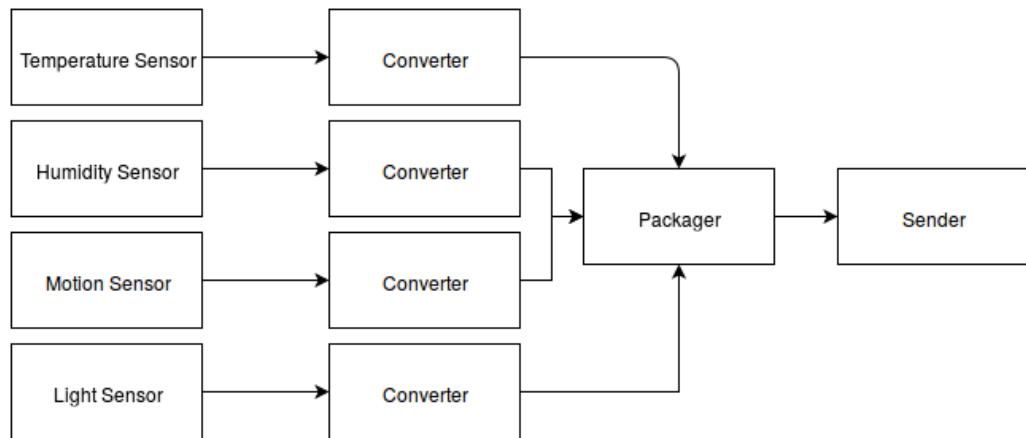


Fig. 18. Diagramma di flusso, per sensore, dei dati

Di seguito si indica un primo marable diagram che mostra le varie trasformazioni che verranno applicate. Questo serve soprattutto per iniziare a capire poi come interpretare eventuali altri schemi come questo.

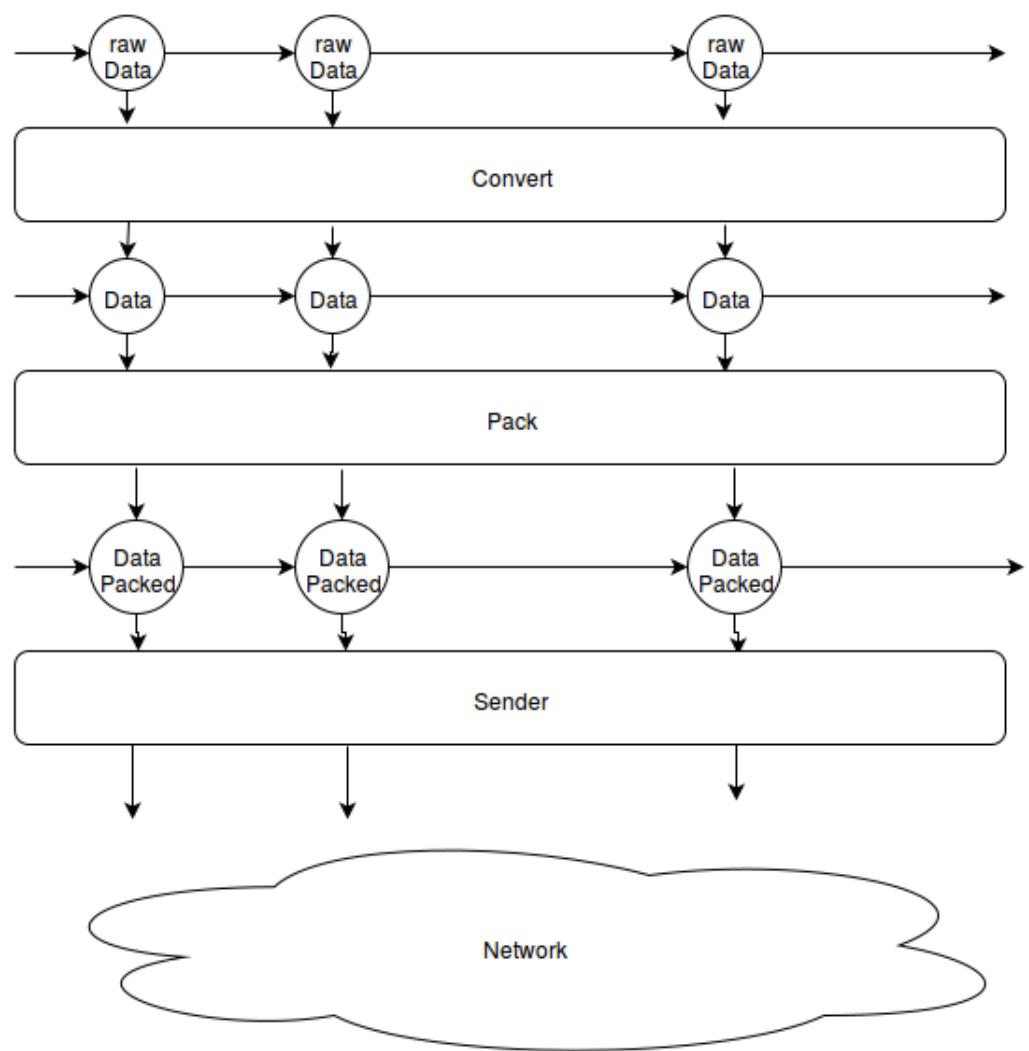


Fig. 19. Marable diagram dei singoli dati proveniente da una sorgente

Struttura

Chiaramente la struttura, che si basa sul precedente step di analisi dei requisiti, risulta cambiata anche alla luce delle assunzioni effettuate e quindi si possono notare l'introduzione di alcune entità che servono per la gestione del sistema embedded e per impostare i flussi di dati provenienti dai sensori.

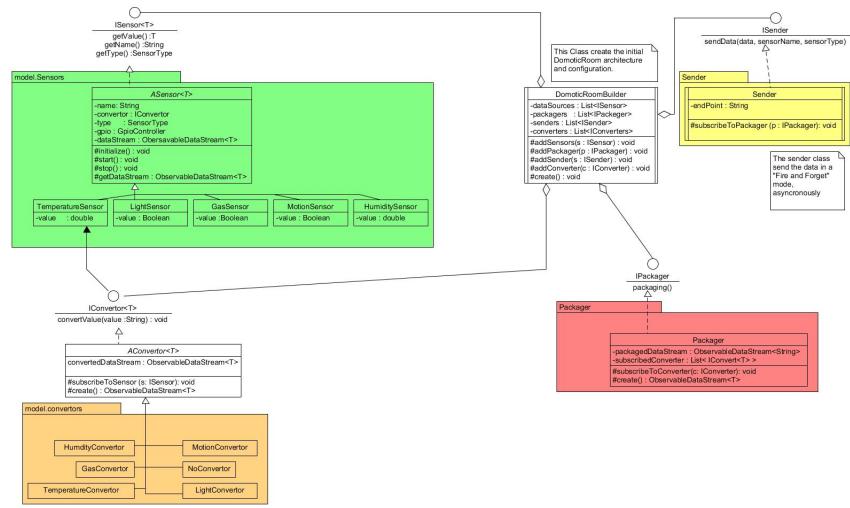


Fig. 20. Logic Architecture, Structure

Come si mostra nello schema soprastante, la struttura logica non ne esce particolarmente modificata rispetto all'analisi iniziale.

Il *SensorDataFetcher* è stato eliminato, poiché l'implementazione di polling per ricevere dati dai sensori viene sostituita dal pattern Observable, utilizzato per sfruttare tutte le potenzialità del paradigma Reactive.

La nuova entità *DomoticRoomBuilder* ha il ruolo centrale di inizializzare il sistema, creando l'architettura e la configurazione iniziale dei suoi componenti.

L'entità *Sensor* ha il ruolo centrale di ricevere i dati dai sensori e inviarli su uno stream dati osservabile. L'entità *Convertor* si occupa di convertire i dati che riceve dalla sorgente *Sensor* a cui è sottoscritto; inoltre genera un nuovo Stream dove invia i dati convertiti che riceve dai *Sensor* a cui è sottoscritto.

L'entità *Packager* unisce tutti i flussi in entrata, dai *Converter* in un unico flusso, il suo compito è preparare i dati ad essere inviati al Server.

L'entità *Sender* si occupa di inviare i dati che riceve dal Packager sulla rete, come pensato nella struttura iniziale.

Interazione

La parte di interazione viene drasticamente modificata proprio perché l'assunzione del paradigma reactive risulta principalmente improntato su questo, in particolare è possibile condensare in meno codice, e più dichiarativo. Vengono in ogni caso mostrati le varie operazioni effettuate sullo stream per mantenere il contatto con quanto mostrato attraverso i marable diagrams. Particolare attenzione va posta poi sulla parte che converte da *procedure calls* a *reactive streams*.

Il sistema non ha una interazione continua; come invece suggerisce il paradigma, esso "reagisce" al verificarsi di un evento di particolare interesse.

In questo caso, il sistema reagisce autonomamente alla ricezione di un valore inviato dal sensore. L'entità *Sensor* si occupa di incanalare queste letture periodiche in un flusso potenzialmente infinito di valori.

L'entità *Convertor* è a sua volta interessata all'arrivo di un nuovo valore sul flusso creato dal *Sensor*; il Convertor applicherà ad ogni nuovo dato una funzione atta a convertire in valore in un dato di maggiore utilità per il sistema.

L'entità *Packager* sarà invece di ascolto sui flussi generati da ogni *Convertor*, appena un nuovo valore convertito è inviato sul flusso; a questo punto il suo compito è "impacchettare" il dato, secondo un protocollo di trasmissione per poterlo inviare al Server.

Quest'ultima parte è gestita dal *Sender* che appena un dato impacchettato è pronto lo invierà al Server.

L'iterazione delle entità del server è ben rappresentata dal diagramma di flusso precedentemente esposto.

Comportamento

Il comportamento atteso dall'Embedded System è molto semplice: una entità DomoticRoomBuilder tramite la funzione addSensor() viene creata l'architettura di iterazione sono esposta, terminata l'operazione è possibile

iniziare la fase di monitoraggio che prevede la lettura dei dati proveniente dai sensori.

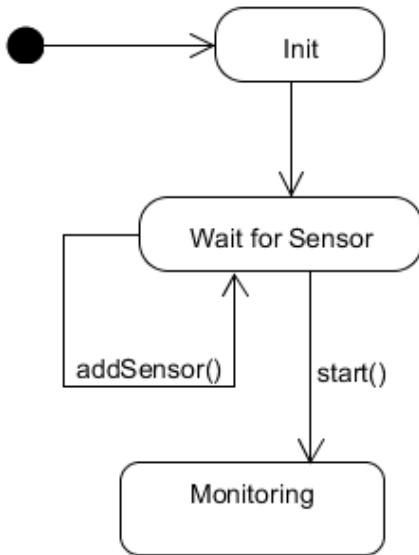


Fig. 21. Diagramma di comportamento

7.1.2 Server

Interazione e Flussi Dati

Anche per quanto riguarda la parte server è necessario definire gli stream che saranno presenti all'interno del sistema. In quanto questa parte è notevolmente più corposa rispetto a quella dell'embedded system per varie ragioni, prima tra tutte la capacità di calcolo, saranno necessari più stream, in particolare si è pensato di creare uno stream per ogni funzionalità sulla base delle operazioni che quindi sono da compiere.

Il primo flusso che andremo ad analizzare riguarda la ricezione dei dati da parte del sistema embedded. In particolare si vuole cercare di riutilizzare le entità che si sono introdotte nell'analisi dei requisiti proprio perché queste

sono maggiormente collegate al dominio. Si può inoltre notare altre entità chiamate *IRawDataFormatter* e *IDBDataFormatter*, queste entità sono utili per preparare la corretta formattazione dei dati e la loro validazione, provenienti dal flusso e dalla successiva elaborazione per essere salvati poi sul database. Si è deciso in particolare di inserirlo per andare a separare i compiti il più possibile e per facilitare la modifica nel caso si voglia adottare uno standard dei dati interno rispetto a quello del client in modo da effettuare una netta indipendenza tra server e client. Soprattutto perché in un'eventuale futuro si può facilmente immaginare un'eterogeneità dei dati dovuti ai vari client disponibili e dalla loro provenienza (altre aziende, protocolli proprietari ...)

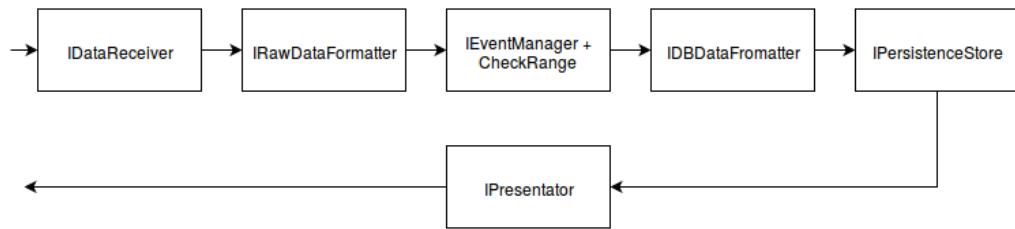


Fig. 22. Server, Flusso Dati Ricezione

Il secondo e ultimo flusso di dati che verrà trattato riguardano le richieste di dati stessi per la visualizzazione da parte dell'utente. Probabilmente per quanto riguarda le seguenti operazioni si poteva anche pensare di gestire il tutto attraverso delle semplici chiamate a procedura, ma, per mantenere una certa coerenza, per sfruttare la dichiaratività del paradigma reactive e per evitare di incorrere nel problema noto come *Callback Hell* si è scelto di utilizzare comunque un flusso. Un'ulteriore vantaggio di gestire attraverso l'infrastruttura reactive consiste nella possibilità di aggiornare realtime l'interfaccia ogni qualvolta avviene l'inserimento di un nuovo valore. Concludendo si vuole far notare la presenza di un'entità che entra in campo quando viene richiesta un'analisi sui dati e non semplicemente una sua visualizzazione. La forma a nuvola indica proprio la possibilità che questo oggetto entri a far parte o meno nel flusso.

Restanti Interazioni

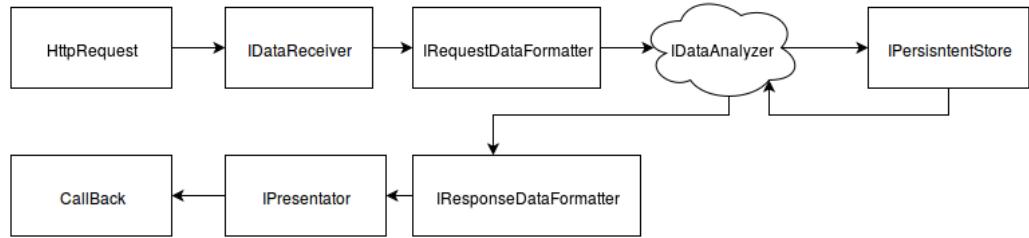


Fig. 23. Server, Flusso Dati Richieste Dati

Per quanto riguarda riguarda l'aggiornamento del range da parte dell'utente, in particolare anche in questo caso l'IEventManager viene chiamato in causa il quale si occuperà di notificare il cambiamento sia a livello di database che al componente incaricato di controllare il range. Questa parte non subisce particolari cambiamenti rispetto a quella dell'analisi dei requisiti perché rimane molto coerente anche a fronte del cambio di paradigma. Un cambiamento significativo riguarda il fatto che il cambiamento del range non avviene più direttamente dall'event manager ma dal database, questo per enfatizzare che il cambiamento del range non è effettivo se non viene prima registrato in maniera permanente.

Questa interazione non avviene attraverso uno stream ma viene gestito come una chiamata classica http in quanto non richiede che ci sia un dialogo continuo tra le varie parti, client e server. Si riporta in ogni caso sotto forma di flusso di chiamate, quelle che vengono effettuate dall'applicativo.

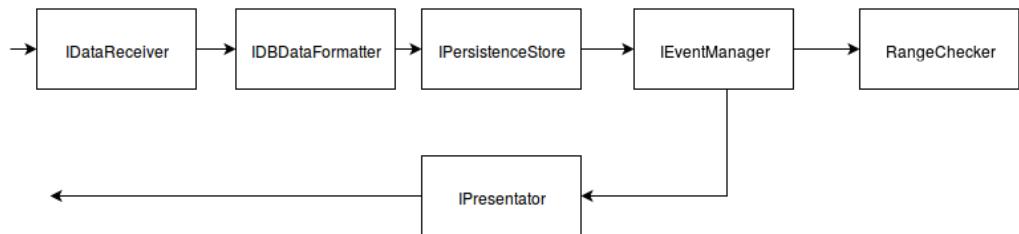


Fig. 24. Server, Flusso Dati Nuovo Range

Prima di passare alla struttura è secondo noi utile andare a definire l'interazione del configurator in quanto è l'unica entità attiva che non si basa effettivamente sui flussi e che contiene l'entry point di tutto il server.

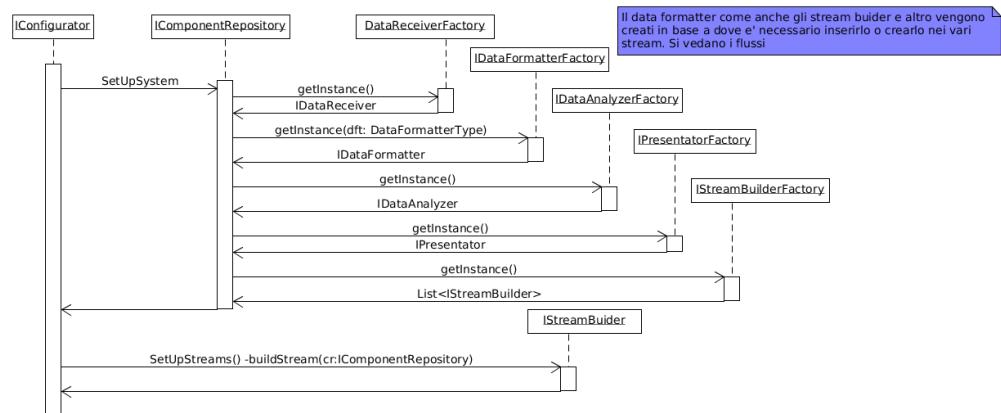


Fig. 25. Server, IConfigurator Interaction

Struttura

Nella struttura dell'analisi del problema abbiamo varie differenze, anche a livello di interfaccia, dovute al cambio di paradigma o ad altre motivazioni, di seguito verranno elencate per rendere tutto più chiaro:

- IDataReceiver: non esiste più il metodo *Receive* perché non si ipotizza più un'approccio a polling e quindi non è necessaria questa funzionalità, mentre invece viene esposto un metodo che da la possibilità di ottenere lo stream di dati associato all'arrivo di un nuovo valore dall'esterno.
- Tutto ciò che in precedenza, nel diagramma dell'interazione, era model-lato attraverso una chiamata a procedura e veniva effettuata all'arrivo di un nuovo dato, ora viene convertito in funzioni che prendono in ingresso un dato stream e lo modificano restituendo un nuovo stream all'inizio del sistema in modo che, in questa fase, si è in grado di costruire e comporre i vari flussi che poi verranno elaborati attraverso l'infrastruttura.
- Per separare ancora di più i vari flussi di stream indicati precedentemente e per separare i compiti di creazione dei vari stream si sono ideate delle classi *factories* che assembleranno i vari componenti a tale scopo.
- Sono stati impostati come oggetti singleton i componenti *EventManager* e *PersistenceStore*
- È stato ampliato la parte del persistenceStore in modo da differenziare i vari compiti di salvataggio e caricamento attraverso diversi oggetti.
- Il DataReceiver è stato differenziato tra dati provenienti dai sensori e dati provenienti dall'utente, come le richieste di un nuovo range o la richiesta di dati. Questo ha di fatto portato all'eliminazione dell'oggetto presentator individuato precedentemente.

In the packages there's only the classes, all the interfaces are in a separate package called "interfaces"

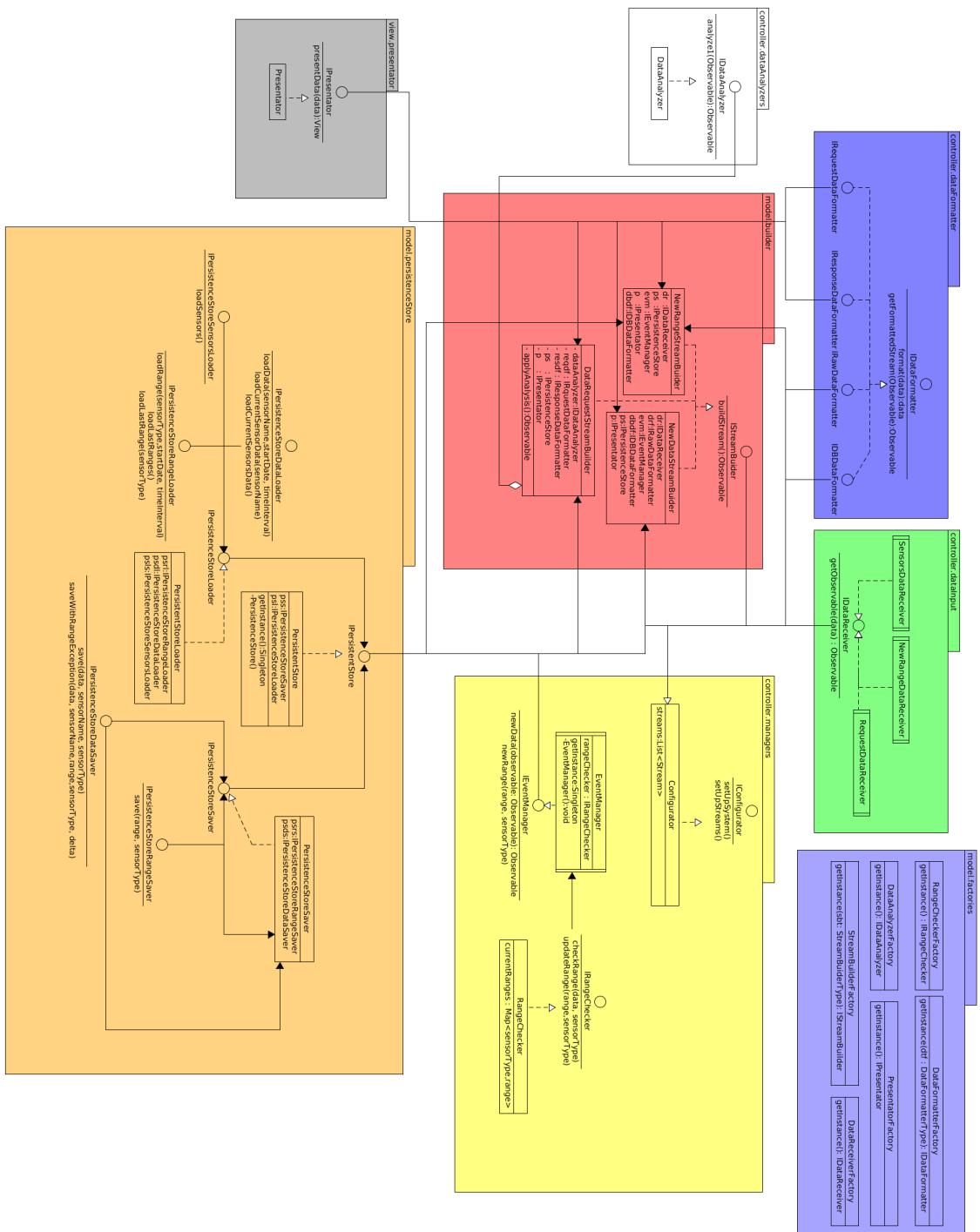


Fig. 26. Server, Struttura Architettura Logica

Comportamento

Per quanto riguarda il comportamento delle varie parti del sistema verranno illustrate quelle principali e descritti gli stati principali incui questi componenti verranno a trovarsi. In particolare verranno omessi tutti quei componenti che effettivamente si occupano solamente della trasformazione degli stream in quanto non posseggono effettivamente uno stato, ma consistono in degli algoritmi che, dato un flusso in ingresso, gli applicano delle apposite trasformazioni e restituiscono un flusso in uscita.

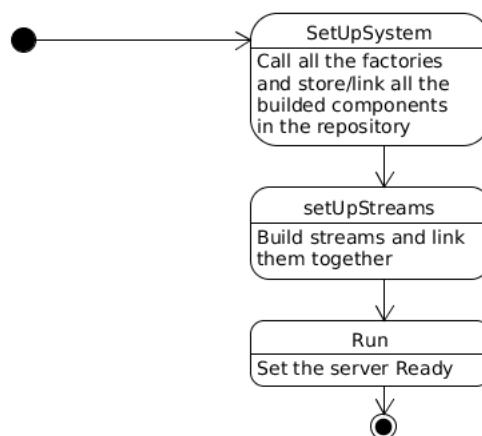


Fig. 27. Server, IConfigurator Behavior

Come si può vedere dalla figura sovrastante il componente IConfigurator sarà quello che si occuperà di attivare tutto il sistema server occupandosi di rendere tutto operativo e pronto per le richieste da soddisfare. Questo avviene appunto in fasi, incui prima viene settato il tutto attraverso una fase di creazione di tutti i componenti necessari, poi avviene la creazione dei flussi che si occuperanno del flow dei dati ed infine si attiveranno questi flussi in modo che siano attivi.

Il comportamento del persistence store rimane molto semplice: una volta creato, attende la richiesta da parte del sistema per la memorizzazione dei dati. Quando una richiesta avviene allora si attiva una delle parti che si occupano di leggere o scrivere all'interno del persistence store e effettuano

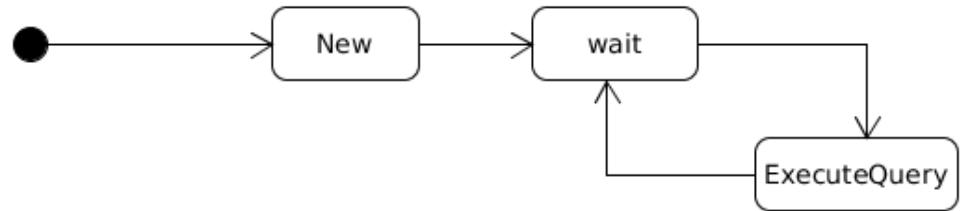


Fig. 28. Server, IPersistenceStore Behavior

l'operazione, per poi tornare in attesa di ulteriori operazioni. Si aggiunge inoltre che questo componente come l'eventManager sono componenti singleton, e quindi è presente solamente un'istanza di questo in tutto il sistema.

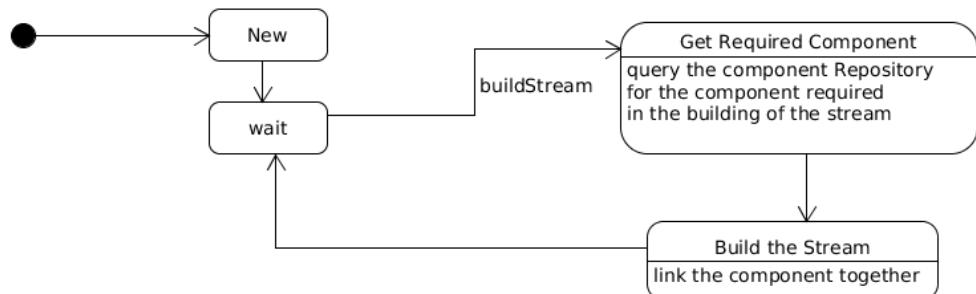


Fig. 29. Server, IStreamBuilder Behavior

Lo stream builder anch'esso risulta molto semplice perché viene chiamato da altri componenti, nel suo caso dal configurator stesso, e per questo presenta due stati come il caso precedente, *New* e *Wait*. All'arrivo di una richiesta il componente richiede i vari blocchi necessari per la costruzione dello stream direttamente dal componentRepository e, una volta ottenuti li collega assieme per costruire il flusso che poi restituirà al configurator stesso.

7.2 Gap di Astrazione

In questa sezione aggiungeremo tutti i tipi di astrazioni richiesti per affrontare il progetto e che non sono direttamente fruibili attraverso la tecnologia di riferimento.

1. Web Server: Data la necessità di comunicare attraverso la rete è necessario che si utilizzi un paradigma a message-passing o attraverso chiamate asincrone, soprattutto per la comunicazione che avviene tra il raspberry e il server.
2. Continuous Integration, Testing and collaborative source control: Lavorando in gruppo sullo stesso repository è necessario impostare il lavoro affinché sia possibile effettuare modifiche in maniera indipendente gli uni dagli altri e allo stesso modo sia possibile controllare automaticamente che i test predisposti e le modifiche effettuate siano coerenti con le specifiche e che il building del progetto sia in ogni caso garantito.
3. Paradigmi Eterogenei: si è deciso di utilizzare dei paradigmi diversi dal OOP classico e questo può portare a problematiche di utilizzo per via dell'inesperienza. Tuttavia queste non vengono fornite direttamente dal linguaggio e quindi si necessitano soluzioni al problema.

7.3 Analisi dei Rischi

In questa sezione verranno elencati i rischi che si potranno incontrare in un progetto di questo tipo formalizzandoli fin da subito, prima di eseguire l'effettiva realizzazione dello stesso, in modo da poterli affrontare e discutere preventivamente per essere pronti nel caso questi si verifichino.

- Aumento dei Costi: è necessario porre particolare attenzione alla struttura hardware del sistema e di come i singoli componenti vanno ad interconnettersi assieme per evitare che si debbano affrontare dei costi aggiuntivi, a progetto già avviato, causati da una qualche mancanza o per via di un'estensione del progetto in corso d'opera.
- Quantitativo della memoria: l'adozione di un database NoSQL porta con sé un certo livello di ridondanza e quindi questo può portare ad un aumento di utilizzo della memoria e di spazio. Il tutto va valutato accuratamente durante il progetto, magari attraverso una serie di prove in base a come è strettamente il dato inizialmente. Se il rischio quindi è reale e quanto è critico.

- Integrazione tra le tecnologie: un possibile rischio riguarda la difficoltà nell'integrare tutte le tecnologie che devono essere sfruttate nel progetto per riuscire a colmare l'abstraction gap e quindi riuscire a completare il progetto attraverso l'analisi appena completata.

8 Work Plan

Il piano di lavoro che abbiamo intenzione di attuare si divide in varie fasi:

1. Controllo del Funzionamento Hardware: la prima cosa che è necessario fare è quella di controllare che tutta la sensoristica sia effettivamente compatibile e che la parte sistemistica sia assemblabile e funzionante.
2. Ricerca e Analisi Problematiche: Viste le problematiche sollevate precedentemente nell'abstraction gap è necessario soffermarsi prima di partire con il progetto per andare ad individuare i tools che possono coprire queste mancanze e quindi avere un'impatto significativo sulla realizzazione del progetto stesso. Queste possono cambiare drasticamente anche la realizzazione del progetto stesso che però deve rimanere fedele all'analisi del problema.
3. Modello del Progetto: costruzione del modello del progetto alla luce delle considerazioni precedenti
4. Impostazione dell'Environment: setup di tutti i tools precedenti e controllo del loro corretto funzionamento
5. Suddivisione dei Compiti: quando tutto è formalizzato adeguatamente è possibile suddividere i vari compiti e quindi parallelizzare il lavoro
6. Cicli di Feedback: è molto utile ai fini della realizzazione del progetto, organizzare dei meeting periodici al fine di effettuare un check sullo stato dei lavori e quindi controllare eventuali incongruenze, discutere i problemi, rivedere i modelli precedenti e altro
7. Integrazione: Integrare tutti i progetti assieme per costruire tutto il sistema assicurandosi che sia corretta l'interazione tra le parti
8. Testing: La parte di testing deve essere sviluppata assieme al codice stesso se possibile sulla base del modello in modo da arrivare alle fasi finali da poter automaticamente capire cosa funziona e cosa no.
9. Deploy: Per la parte di deploy non si porrà particolare attenzione in questa prima versione dal momento in cui non è effettivamente richiesta una particolare complessità nel deploy su più macchine. Comunque

si tratta di un'aspetto che può essere affrontato anche a progetto finito nel quale si potrebbe procedere ad apportare le modifiche richieste per realizzare un deploy più articolato.

8.1 Strumenti e Framework

In questa sottosezione vengono illustrati i tools utilizzati durante questo progetto utili a cercare di colmare l'abstraction gap e per il supporto alla gestione del progetto stesso.

- Umlet: Questo tool è stato utilizzato per creare gli schemi UML che realizzano i modelli di tutto il progetto.[?]
- Play Framework and Akka: framework creato da Typesafe che consente di gestire un web server con REST API e di gestire autonomamente le chiamate in maniera asincrona. Akka è tutta l'infrastruttura sottostante che consente di gestire tutto questo. Per ulteriori informazioni è sufficiente cercare nel web.[?,?]
- Activator Template: tool associato al framework precedente che consente di gestire le proprie applicazioni secondo degli standard affermati e delle configurazioni di default in modo da velocizzare lo startup di tutto il progetto.[?]
- Scala Build Tool: anche se è stato pensato per progetti in scala questo strumento funziona correttamente anche per java e consente di gestire tutte le dipendenze, eventuali librerie aggiuntive e configurazioni. Molti dei progetti precedenti vengono inseriti all'interno del sistema attraverso questo.[?]
- Bootstrap: Si tratta di una libreria grafica per gestire il frontend e renderlo più piacevole....
- MongoDB: Al fine di coprire alcuni temi del corso si è deciso di adottare un database NoSQL, in particolare perché questo, attraverso una rappresentazione a documenti e del tutto simile al formato JSON, fortemente utilizzato in ambito web, risulta molto adatto al problema. Inoltre ci consente di memorizzare i dati in maniera dinamica in modo che, se in un futuro l'applicazione dovesse ingrandirsi o se devono essere fatte delle aggiunte/modifiche, queste possano essere fatte in agilmente. Infine, l'ultimo vantaggio consiste nel memorizzare i dati esattamente come possono essere utili al sistema, evitando il prezzo delle join relazionali. [?]

- reactiveMongo: Per l’accesso al database si è deciso di utilizzare una libreria che meglio incarna l’idea di stream effettuando appunto accessi completamente asincroni e quindi rendendo il tutto non bloccante, in piena idea reattiva.[?]
- c3.js: Libreria javascript per la rappresentazione dei dati sul web. [?]
- Scalacheck: Libreria scala per la realizzazzione di *property based testing* [?]
- RxPY: libreria che consente di utilizzare il paradigma reactive in tecnologia Python [?]
- RxJs: libreria che consente di utilizzare il paradigma reactive in tecnologia Javascript, quindi per gestire stream di dati provenienti dal server [?]

9 Project

9.1 Database e Altre Rappresentazioni di Dati

Per la gestione del database si è scelto di utilizzare un database NoSql, nell’accezione MongoDB[?]. In questa sede non si discuteranno i vantaggi, gli svantaggi o le particolarità di questa tecnologia in quanto è stato fatto già ampiamente durante il corso. Di seguito si riportano gli schemi dei documenti che sono stati salvati, le collezioni e il significato di alcuni dei valori inseriti.

Collection Ranges

In questa collezione di documenti vengono raggruppati tutti i documenti relativi ai ranges che sono da controllare. In particolare sono stati individuati due tipologie di range in base alla tipologia di sensori.

- Si/No: Per quella tipologia di sensori che non forniscono effettivamente dei valori ma che notificano solamente la presenza o l’assenza di un particolare elemento come il GAS o il movimento.
- Valore: validi quando un sensore effettivamente serve una misurazione di una grandezza, come ad esempio la temperatura. In questo caso è utile sapere se questa grandezza rimane all’interno di determinati range.

```

1  {
2      "_id": ObjectId("56fb3cc3a0000650b4ec067"),
3      "value": true,
4      "rangeType": 3,
5      "dateCreated": ISODate("2016-03-30T13:25:32.713Z")
6  }

```

Fig. 30. Documento dei Ranges Booleani

```

1  {
2      "_id": ObjectId("56fb3b53a00003c004ec066"),
3      "minBound": 0,
4      "maxBound": 20,
5      "rangeType": 5,
6      "dateCreated": ISODate("2016-03-30T13:25:09.821Z")
7  }

```

Fig. 31. Documento dei Ranges di Valori

Si vuole far notare come sia comunque sempre presente un attributo di tipo data in modo da poter filtrare i range in base al tempo di inserimento e la presenza di un valore numerico che in questo caso rappresenta il tipo di range.

Per indicare il tipo di range si riporta il listato scala che indica tale tipologia.

Collertion Sensors

Per quanto riguarda la collezione dei sensori si utilizza un'apposita collection. Questa non sarà modificata spesso, e serve principalmente per eventuali evoluzioni del progetto, in modo che sia già presente. Anch'essi sono dotati di di tipo come i range e, per questa implementazione, i tipi coincidono. A livello di progetto però si è deciso di mantenerli separati nel caso in un futuro questi divergano. Inseriamo di seguito un'immagine che rappresenta come è strutturato un documento di un sensore.

```

///////////////////////////////
// SENSOR TYPE TO - FROM
/////////////////////////////
// RangeType
////////////////////////////

def sensorTypeToRangeType(sensorType : SensorType): RangeType = sensorType match {
  case SensorType.Gas          => RangeType.Gas
  case SensorType.Humidity     => RangeType.Humidity
  case SensorType.Light         => RangeType.Light
  case SensorType.Movement      => RangeType.Movement
  case SensorType.Temperature  => RangeType.Temperature
}

/////////////////////////////
// INTEGER
////////////////////////////

def intToSensorType(id : Int): SensorType = id match {
  case 1 => SensorType.Gas
  case 2 => SensorType.Humidity
  case 3 => SensorType.Light
  case 4 => SensorType.Movement
  case 5 => SensorType.Temperature
}

def sensorTypeToInt(sensorType: SensorType) = sensorType match {
  case SensorType.Gas          => 1
  case SensorType.Humidity     => 2
  case SensorType.Light         => 3
  case SensorType.Movement      => 4
  case SensorType.Temperature  => 5
}

```

Fig. 32. Valore Numerico dei Ranges

```

1 | {
2 |   "_id": ObjectId("56a3f9d590b5e01b53de6389"),
3 |   "sensorName": "lightSensor",
4 |   "type": 2
5 | }

```

Fig. 33. Rappresentazione a DB dei Sensori

```
//////////  
// INTEGER  
//////////  
  
def intToSensorType(id : Int): SensorType = id match {  
    case 1 => SensorType.Gas  
    case 2 => SensorType.Humidity  
    case 3 => SensorType.Light  
    case 4 => SensorType.Movement  
    case 5 => SensorType.Temperature  
}  
  
def sensorTypeToInt(sensorType: SensorType) = sensorType match {  
    case SensorType.Gas        => 1  
    case SensorType.Humidity   => 2  
    case SensorType.Light      => 3  
    case SensorType.Movement   => 4  
    case SensorType.Temperature => 5  
}
```

Fig. 34. Valore Numerico dei Sensori

Collection Data

Per quanto riguarda la rappresentazione dei dati raccolti, anche in questo caso sono presenti 2 tipologie di dati, che a loro volta differiscono leggermente per la tipologia di dato, se booleano o meno:

- Dati Corretti: cioè quella tipologia di dati che rientrano all'interno del range prestabilito per la loro tipologia.
- Dati Incorretti: quelli che invece non rientrano nella tipologia corretta e quindi vanno a violare il range attivo.

Come si può osservare il primo tipo è più semplice in quanto non richiede di salvare anche l'informazione riguardante di quanto il dato risulta errato. Di seguito come fatto in precedenza si indica un esempio di struttura di un dato corretto e non corretto per ogni tipologia di dato.

```
1 {  
2   "_id": ObjectId("5703fb714700005800967847"),  
3   "dateCreation": ISODate("2016-04-05T15:52:49.435Z"),  
4   "sensorName": "nonBooleanSensorName5",  
5   "type": 5,  
6   "value": 4  
7 }
```

Fig. 35. Dato Non Booleano Corretto salvato in database

Come si può osservare non è stato comunque inserito un valore per la violazione in modo da poter distinguere meglio i dati violati da quelli invece corretti.

```
1 {  
2   "_id": ObjectId("5703fb6c470000580096783d"),  
3   "dateCreation": ISODate("2016-04-05T15:52:44.305Z"),  
4   "rangeViolation": {  
5     "delta": 60  
6   },  
7   "sensorName": "nonBooleanSensorName5",  
8   "type": 5,  
9   "value": 80  
10 }
```

Fig. 36. Dato Non Booleano Non Corretto salvato in database

In questo caso appunto il valore della violazione non e' piu' nullo ma contiene un semplice campo *delta* che indicherà la differenza rispetto al range. In particolare questo sarà positivo se il valore registrato supera il range attuale, negativo se invece è troppo basso o zero se il tipo di sensoristica e quindi anche di range è booleano. Come si puo' vedere nelle seguenti immagini.

```
1 {  
2   "_id": ObjectId("5703fb6c470000580096783c"),  
3   "dateCreation": ISODate("2016-04-05T15:52:43.793Z"),  
4   "sensorName": "booleanSensorName4",  
5   "type": 4,  
6   "value": false  
7 }
```

Fig. 37. Dato Booleano Corretto salvato in database

```
1 {  
2   "_id": ObjectId("5703fb6c470000570096783e"),  
3   "dateCreation": ISODate("2016-04-05T15:52:44.819Z"),  
4   "rangeViolation": {  
5     "delta": 0  
6   },  
7   "sensorName": "booleanSensorName3",  
8   "type": 3,  
9   "value": false  
10 }
```

Fig. 38. Dato Non Booleano Non Corretto salvato in database

Rappresentazione Dati Attraverso la Rete

Oltre alla rappresentazione dei dati per quanto riguarda i dati salvati all'interno del Database è stato necessario concordare una rappresentazione di dati che verranno inviati attraverso la rete. Non necessariamente le due rappresentazioni devono coincidere in quanto, i dispositivi dai quali si ricavano i dati posso avere diversi formati e modalità di ottenere tali valori, quindi di conseguenza questa rappresentazione è sbilanciata rispetto a chi invia i dati, cioè il sistema embedded.

Anche per la rappresentazione dei dati si è scelta la rappresentazione JSON in quanto questa è una delle più utilizzate nella comunicazione eterogenea via rete, si pone quindi come formato interoperabile tra varie tecnologie, come accade nel nostro caso.

Di seguito si riporta un esempio di dati in formato JSON:

```
1 {  
2     "sensorName": "nomeSensore",  
3     "sensorType": 1,  
4     "value": true,  
5     "date": "2016-04-05 17:52:55.058"  
6 }
```

Si veda il tipo di sensore dai valori indicati precedentemente e si sottolinea che il tipo del campo *value* può essere sia booleano che numerico.

9.2 Introduzione All'Architettura di Progetto

Nelle prossime sezioni si vuole inserire le modifiche che sono state fatte alla architettura logica appena introdotta, ma cercando di evitare di ripetere aspetti che non sono effettivamente cambiati. Di conseguenza si è deciso di riportare solamente quegli aspetti che sono stati modificati in base alle scelte tecnologiche effettivamente fatte.

In ogni caso si è deciso di cercare il più possibile di sfruttare quanto già è stato concordato dagli analisti cercando di effettuare meno modifiche possibili all'architettura, ma allo stesso tempo di cercare di ridurre le entità che, per ragioni tecnologiche, risultano inutili per la creazione del progetto.

Server

9.3 Struttura

Vista l'introduzione della tecnologia in questa fase si è deciso di apportare alcune modifiche architetturali. Si riporta una lista di modifiche apportate all'architettura:

- *DataStructures*: Si è ritenuto necessario introdurre un'ulteriore package che non era stato previsto precedentemente contenente le rappresentazioni interne in forma di classi dei dati che circolano nel sistema, così come delle classi di utilità che servono per la conversione, da e verso, questi dati. (eg . JSON, BSON, DATETIME)
- *IDataFormatter*: Visto che le elaborazioni di trasformazione dei dati non richiedono tanta computazione si è deciso di eliminare il metodo che, dato un valore lo validava inglobandolo direttamente dentro il primo metodo che restituisce lo stream. Inoltre nella costruzione dello stream ci si è accorti che il valore dell'input non viene passato per parametro al metodo, ma viene gestito dall'infrastruttura, quindi si è deciso di eliminarlo. Spesso gli oggetti che validano e formattano i dati passano da una rappresentazione ad un'altra: tra valori JSON, BSON e classi interne per i dati.
- *IDBDataFormatter*: Viene aggiunto un'ulteriore metodo che gestisce la creazione di dati da un formato inherente alle strutture dati presenti in un altro utilizzabile per lo storage dei dati nel DB. Questo metodo è estraneo a quelli per la costruzione di stream e serve per la gestione delle richieste di inserimento di un nuovo range.
- *Eliminazione della Maggior Parte dei Formatter*: Questa scelta è data dalla possibilità che PlayFramework e altre build in librerie che abbiamo utilizzato consentono per la conversione automatica dei dati da un formato ad un altro, di conseguenza queste operazioni vengono agilmente svolte all'interno del package DataStructures.
- *Eliminazione del Presentator*: Anche in questo caso le funzionalità di playframework ci hanno evitato questa implementazione.
- *Eliminazione delle Factory*: Visto che la maggior parte delle classi che erano state pensate non inglobano uno stato, ma vengono utilizzate semplicemente per applicare delle computazioni allo stream allora sono state eliminate la maggior parte delle factory class. Grazie anche alla tecnologia scelta (scala) molte delle classi sono state convertite in objects (build-in singletons).

- *Eliminazione della classe Configurator:* Si è deciso di eliminare tale classe in quanto non è più possibile pensare a un entry point per l'applicazione, ma questa viene risvegliata attraverso chiamate rest e di conseguenza molti dei processi legati alla creazione di oggetti è stata ridotta.
- *Eliminazione del Data Analysis:* A fronte delle semplici operazioni di analisi sui dati (Min, Max, Average) si è deciso di sfruttare direttamente le capacità forniteci dal database.
- *StreamBuilder:* Le classi che implementano gli stream builder sono state leggermente modificate. In particolare si è previsto uno stream builder per la creazione dello stream riguardante i dati, uno per il salvataggio dei dati e uno per la gestione dei range. Si vedano poi lo schema della struttura e interazione.

Si di seguito riporta quindi lo schema della struttura di progetto. Purtroppo però alcune delle astrazioni presenti nel linguaggio scala non possono essere completamente catturate dal linguaggio di modellazione UML. Per rendere poi lo schema più pulito si è deciso di omettere tutte le implementazioni che non aggiungono valore alle interfacce.

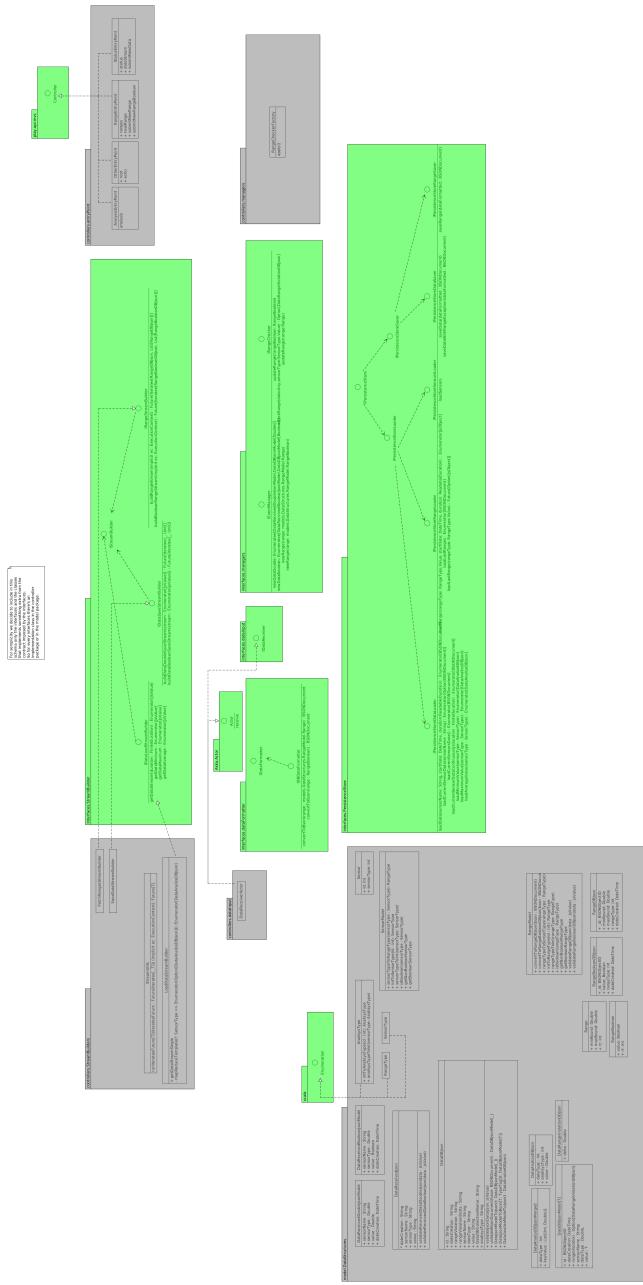


Fig. 39. Struttura di progetto del Server

9.4 Interazione

In questa parte si riporta la tecnologia che si è utilizzata per effettuare l’interazione che si è analizzata e indicata in fase di analisi, lo stream. In particolare si è notato che, il framework play[?] già gestisce attraverso delle sue strutture la possibilità di operare attraverso stream di dati. Questa possibilità è stata introdotta dal framework per gestire meglio dati parziali o casi che richiederebbero molto tempo, come ad esempio un’upload di file cospicuo. Quindi ci avvarremo di questa astrazione per implementare quanto descritto nella fase di analisi del problema. Si riporta in bibliografia quindi la documentazione del framework che spiega il funzionamento di *Enumerators, Iteratees e Enumeratees*. [?]

Un’ulteriore modifica che va apportata visto che ci si trova in ambito web e si dispone di questo framework è la necessità di eliminare o aggiungere alcune entità previste precedentemente perché, in fase realizzativa sono risultate ridondanti, ad esempio:

- **DataInputs:** Viene rimpiazzato dai controller che gestiscono direttamente le richieste POST al sistema per la gestione dei dati provenienti dai form HTTP, o dal sistema embedded. Saranno questi componenti, come ad esempio il *RangeEntryPoint* a validare direttamente i dati in arrivo e a chiamare poi gli elementi successivi nel flusso di dati per il salvataggio di un nuovo range. Si veda l’analisi del problema.
- **DataReceiverActor:** Attore basato sul framework akka che consente di gestire lo stream di input, in particolare consente di aggiungere incrementalmente dati all stream mano a mano che questi arrivano senza che sia necessario ricrearlo.

Infine, visto che ci si trova in fase di progetto e che gli schemi principali sono già stati inseriti in fase di analisi, in questo punto si è deciso di inserire degli snippet di codice come schema di interazione in quanto, essendo provenienti dalla programmazione reactive risultano molto dichiarativi e quindi espressivi al fine della modellazione.

Stream di Salvataggio Dati: di seguito vengono inseriti gli snippet principali che mostrano il salvataggio dei dati attraverso la costruzione e l’esecuzione di uno stream, in particolare nella prima (fig.40) si può osservare come i dati, provenienti dalla rete vengono controllati inizialmente se sono conformi alla

specifica JSON e poi passati all'attore (fig.41) che costruisce uno stream con un canale incui possono essere inseriti a mano a mano nuovi dati. Infine nell'ultima parte si costruisce effettivamente tutto lo stream attraverso vari steps che consentono di salvare tali dati e controllarne la correttezza. (fig.42)

Stream di Caricamento Dati: Anche questo stream parte chiaramente dal controller in quanto, alla chiamata della pagina di stato dei sensori, viene prima richiesti i dati per il valore dei range (si veda il paragrafo apposito) i quali vengono passati alla vista che li userà per la visualizzazione dei limiti imposti da questi all'interno dei grafici. Successivamente sempre la vista richiederà i dati ad un tempo prefissato, in modo da comandare il flusso e evitare backpressure, attraverso l'utilizzo di una websocket instaurata tra client e server (fig 43). Infine sempre lo streambuilder si occuperà di ottenere i dati attraverso una catena di computazioni come si può vedere in figura 44

Stream di Caricamento Ranges: Per quanto riguarda il caricamento dei ranges, utilizzato anche in figura 43 e nella relativa pagina di visualizzazione dei sensori, si utilizza uno stream che recupera i dati a database attraverso una query di raggruppamento che ottiene solamente i valori più aggiornati. In particolare gli stream sono due distinti sulla base della tipologia dei range in quanto, un tipo mostra i dati all'interno di valori numerici, mentre l'altro si occupa dei valori booleani, si veda la figura 45

Stream di Caricamento Dati Analisi: Anche per il recupero dei dati di analisi si utilizza l'astrazione a stream come si può vedere in figura 46. La cosa notevole di questo stream riguarda l'utilizzo di un pattern map-reduce nella sua realizzazione.

```

def submitNewData = Action { implicit request =>
  val content = request.body.asJson
  content match {
    case Some(x)  => {
      dataReceiverActor ! x
      Ok
    }
    case None     => BadRequest("the content of the request is not a Json Value")
  }
}

```

Fig. 40. Step 1 - Stream di Salvataggio D@ati

```

val (dataEnumerator,dataEnumeratorChannel) = Concurrent.broadcast[JsValue]

def receive = {
  case message: JsValue =>
    // push the message to the stream
    dataEnumeratorChannel.push(message)

    case "getStream" => sender() ! dataEnumerator
  }
}

```

Fig. 41. Step 2 - Stream di Salvataggio Dati

```

object SaveDataStreamBuilder extends IDataSaveStreamBuilder {
  override def buildDataDoubleSaveStream(stream : Enumerator[JsValue]): Future[Iteratee[BSONDocument, Unit]] = {
    (stream &>
      Enumeratee.filter(x => {
        DataReceivedJson.validateReceivedDataDoubleJson(x).isSuccess
      }) &>
      Enumeratee.map(y => DataReceivedJson.validateReceivedDataDoubleJson(y).get) &>
      EventManager newDataDouble &>
      Enumeratee.map(y => DataDBJson.DataJsonModelToBson(y)))(Iteratee.foreach[BSONDocument](z => PersistenceStore.saveData(z)))
  }

  override def buildDataBooleanSaveStream(stream : Enumerator[JsValue]): Future[Iteratee[BSONDocument, Unit]] = {
    (stream &>
      Enumeratee.filter(x => {
        DataReceivedJson.validateReceivedDataBooleanJson(x).isSuccess
      }) &>
      Enumeratee.map(y => DataReceivedJson.validateReceivedDataBooleanJson(y).get) &>
      EventManager newDataBoolean &>
      Enumeratee.map(y => DataDBJson.DataJsonModelToBson(y)))(Iteratee.foreach[BSONDocument](z => PersistenceStore.saveData(z)))
  }
}

```

Fig. 42. Step 3 - Stream di Salvataggio Dati

```

def status = Action {
    val rangesStream = FetchRangesStreamBuilder.buildRangeStream
    val ranges = StreamUtils.runIterateeFuture(rangesStream)

    OK(views.html.statusView.render(Await.result(ranges,30.seconds)))
}

def dataStream = WebSocket.using[JsValue]{ request =>

    // Concurrent.broadcast returns (Enumerator, Concurrent.Channel)
    val (outRequestEnumerator, channel) = Concurrent.broadcast[JsValue]

    //LoadDataStreamBuilder.getDataStream[30.seconds]
    val out : Enumerator[JsValue] = LoadDataStreamBuilder.getDataStreamSingle() >- outRequestEnumerator

    // log the message to stdout and send response back to client
    val in = Iteratee.foreach[JsValue] {
        msg => println(msg)
        // the Enumerator returned by Concurrent.broadcast subscribes to the channel and will
        // receive the pushed messages
        LoadDataStreamBuilder.getDataStreamSingle()(Iteratee.foreach(x =>{
            println(x)
            channel push(x)
        }))
    }(in,out)
}

```

Fig. 43. Step 1 - Stream di Caricamento Dati

```

def getDataStreamSingle() : Enumerator[JsValue] = {
    val stream = (PersistenceStore.loadCurrentSensorsData() &>
        Enumeratee.map(x => DataDBJson.validateBsonDocument(x)) &>
        Enumeratee.map(x => DataDBJson.DataJsonModelToJson(x)))
    stream
}

```

Fig. 44. Step 2 - Stream di Caricamento Dati

```

object FetchRangesStreamBuilder extends IRangeStreamBuilder{
    override def buildRangeStream(implicit ec: ExecutionContext) = (PersistenceStore.loadLastRanges &>
        Enumeratee.filter(x => RangeDBJsonHandler.readTry(x).isSuccess) &>
        Enumeratee.map(y => RangeDBJsonHandler.read(y)))
        .apply(Iteratee.fold[RangeDBJson, List[RangeDBJson]](List.empty) { (s, e) => s :: (e) })

    override def buildBooleanRangeStream(implicit ec: ExecutionContext) = (PersistenceStore.loadLastRanges &>
        Enumeratee.filter(x => RangeBooleanDBJsonHandler.readTry(x).isSuccess) &>
        Enumeratee.map(y => RangeBooleanDBJsonHandler.read(y)))
        .apply(Iteratee.fold[RangeBooleanDBJson, List[RangeBooleanDBJson]](List.empty) { (s, e) => s :: (e) })
}

```

Fig. 45. Stream di Caricamento Ranges

```

def getDataMinimum : Enumerator[DataAnalyzeDBJson] = {
    mapReduceTemplate(PersistenceStore.loadMinimumValue)
}

def getDataMaximum : Enumerator[DataAnalyzeDBJson] = {
    mapReduceTemplate(PersistenceStore.loadMaximumValue)
}

def getDataAverage : Enumerator[DataAnalyzeDBJson] = {
    mapReduceTemplate(PersistenceStore.loadAverageValue)
}

private def mapReduceTemplate(f: SensorType => Enumerator[Option[DataAnalyzeDBJson]]) = {
    val stream : Enumerator[DataAnalyzeDBJson] = SensorModel.getBooleanSensorType map { sensorType =>
        f(sensorType).map[DataAnalyzeDBJson](x => if (x.isDefined) x.get else DataAnalyzeDBJson(-1,6,0))
    } reduce { (x,y) => x.andThen(y)}
    stream
}

```

Fig. 46. Stream di Caricamento Dati di Analysis

9.5 Comportamento

Per quanto concerne il comportamento, verranno mostrati gli stati attraverso i quali il server transita durante il suo funzionamento. Avendo utilizzato un approccio reactive e funzionale dove possibile, il numero di stati collezionati all'interno delle varie entità si riduce. Infatti si è cercato di utilizzare tipi *object* dove possibile, che corrispondono a un singleton pattern, ma solamente che istanziati da parte del framework. In particolare questi oggetti non contengono dello stato.

Solo la classe *RangeChecker* contiene uno stato che corrisponde alla collezione dei range attualmente attivi all'interno del server. Le uniche altre classi che si sono implementate sono quelle che effettivamente costituiscono l'oggetto *PersistenceStore* secondo uno strategy pattern. Si è scelto di lasciarle come classi in quanto possono essere dinamicamente interscambiabili con altri oggetti che possono operare su basi di dati differenti, ma mantenendo lo stesso contratto/interfaccia.

Per quanto concerne la lista di stati necessari tutto si riduce a:

1. **Inizializzazione:** eseguita maggiormente dal sistema che si occupa di creare gli object e quindi instanziare tutte le loro dipendenze in base a quello che viene definito nei loro costruttori. Questa operazione viene effettuata in maniera lazy in modo da enfatizzare la velocità di esecuzione e ridurre l'utilizzo di risorse inutili al momento.
2. **Modifica dei Range:** Alla modifica dei range deve conseguire anche una modifica in memoria della collection che contiene i range attivi al momento, quindi si attiva un processo di aggiornamento nell'oggetto *RangeChecker*. Quando questo processo termina il suo decorso il tutto torna a funzionare sulla base della costruzione degli stream che si è discussa in fase di analisi.

Per quanto riguarda lo schema del comportamento quindi tutto diventa semplicemente:

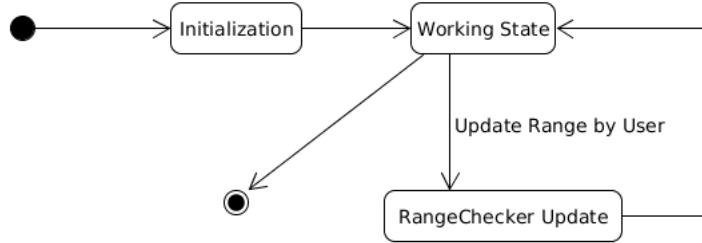


Fig. 47. Schema di Behaviour del server

Sistema Embedded

9.6 Struttura

La struttura del Sistema Embedded non ha subito cambiamenti drastici dalla progettazione logica, sono state solo apportate alcune modifiche necessarie per i limiti e le proprietà imposte dall'Hardware e dalle tecnologie utilizzate. La prima modifica è l'assenza delle Interface, sostituite interamente dalle classi Abstract, questo perché Python non prevede l'implementazione delle Interface. L'entità *ASensor* quindi rappresenta il comportamento comune a tutti i sensori, come l'inizializzazione sulla GPIO Board. La classe *ASensor* poi viene effettivamente implementata dalle classi:

- *TemperatureSensor*: Rappresenta il sensore di Temperatura, il quale dopo un intervallo di tempo determinato all'istanziazione richiede al sensore il valore percepito.
- *BooleanSensor*: I sensori Booleani invece comunicano col Sistema in modo Asincrono notificando solamente quando percepiscono un cambiamento dell'ambiente monitorato.

Si può quindi notare come il sensore di temperatura sia gestito a polling dal sistema, con letture fatte ad intervalli regolari; a differenza dei sensori Booleani che invece vengono gestiti tramite Interupt.

Ogni sensore implementa un canale Stream per l'invio dei dati, come previsto dal progetto logico. I dati inviati nel canale Stream vengono elaborati dalle entità di tipo *AStreamWorker*, le quali vengono poi implementate in differenti specializzazioni, in base al compito a loro affidato:

- *Validator*: Il Validator è l'unica entità dal prevista inizialmente dal progetto logico, è stata aggiunta perché si occupa di validare tramite checksum il valore ricevuto dal sensore di Temperatura.
- *Converter*: Il Converter invece modifica i valori ricevuti dal Validator secondo una particolare funzione di conversione.
- *Packager*: Il Packager trasforma i dati ricevuti dai Convert in una stringa in formato Json che contiene tutte le informazioni necessarie al Server.
- *Sender*: L'ultimo StreamWorker è rappresentato dal Sender che invia la stringa Json tramite richiesta http al Server.

Infine è stata implementata *DomoticRoomBuilder* per l'inizializzazione del Sistema Embedded, la quale inizializza i Sensori e il reticolo di Stream-Worker necessario per la gestione dei valori.

9.7 Interazione

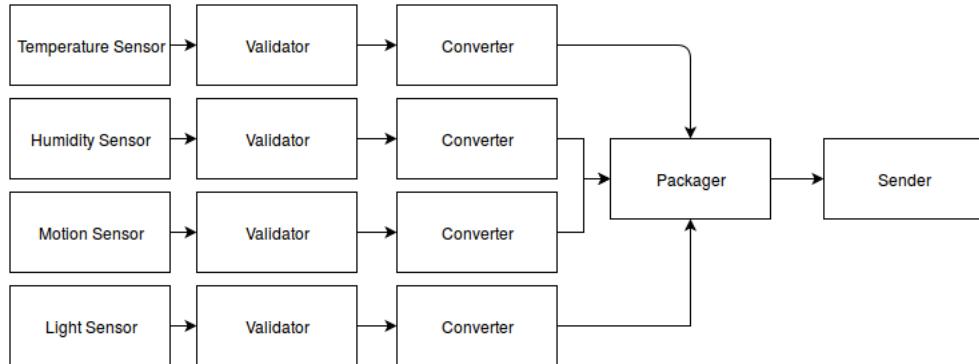


Fig. 48. Marable diagram dei singoli dati proveniente da una sorgente

L'interazione del sistema Embedded è rimasta pressoché invariata da quanto descritto nel progetto logico. L'introduzione dell'entità Validator aggiunge, semplicemente, un altro passaggio al flusso di elaborazione dei dati rilevati dai sensori. Questo dimostra come il flusso di StreamWorker sia una struttura molto flessibile ed estendibile in base alle necessità del sistema.

9.8 Comportamento

Il sistema viene inizializzato dalla classe *DomoticRoomBuilder*, la quale inizializza i sensori e crea gli StreamWorker necessari alla gestione del flusso dei dati. L'unica informazione richiesta all'utente è l'indirizzo IP e la porta tramite cui comunicare col Server.

10 Configurazione Hardware

Dopo aver procurato tutto il materiale utile all'implementazione fisica del progetto è necessario collegare tutto l'hardware in maniera corretta.

In questa fase è molto importante conoscere le uscite dei sensori e ogni singolo pin della Gpio del Raspberry. Per quanto riguarda i pin dei sensori, solitamente la loro funzione è riportata direttamente nel dispositivo. Per quanto riguarda la Gpio del raspberry è bene aver sotto mano lo schema corrispondente che riporta ogni singola funzione di ciascun pin.

10.1 Schema di collegamento

IMPORTANTE: Quando si procede a collegare i sensori al Raspberry è bene assicurarsi che i pin utilizzati siano corretti, soprattutto per quanto riguarda quelli di alimentazione (5v o 3,3v) in quanto uno sbagliato uso di essi può compromettere in maniera permanente il sensore.

Lo schema di collegamento (figura 49) mostra la Gpio del Raspberry e gli accoppiamenti utilizzati nei sensori. Qual'ora il progetto voglia essere esteso sarà necessario procurarsi una fonte di alimentazione esterna in quanto il Raspberry mette a disposizione solo quattro pin di alimentazione.

Raspberry Pi B+
B+ J8 GPIO Header

Pin No.		
3.3V	1	2 5V
GPIO2	3	4 5V
GPIO3	5	6 GND
GPIO4	7	8 GPIO14
GND	9	10 GPIO15
GPIO17	11	12 GPIO18
GPIO27	13	14 GND
GPIO22	15	16 GPIO23
3.3V	17	18 GPIO24
GPIO10	19	20 GND
GPIO9	21	22 GPIO25
GPIO11	23	24 GPIO8
GND	25	26 GPIO7
DNC	27	28 DNC
GPIO5	29	30 GND
GPIO6	31	32 GPIO12
GPIO13	33	34 GND
GPIO19	35	36 GPIO16
GPIO26	37	38 GPIO20
GND	39	40 GPIO21

Fig. 49. Schema di collegamento

11 Implementazione

In questa fase bisognerebbe riportare il codice che implementa le varie parti, però, per non appesantire troppo questa relazione che vuole essere una spiegazione alle scelte progettuali e di anali effettuate, si rimanda al lettore la revisione del codice. Di seguito si inserisce il link al repository contenente tutti i sorgenti del progetto stesso, nonchè il sorgente di tale documentazione.

<https://github.com/benkio/DomoticRoom>

12 Testing

12.1 Server

Per quanto riguarda la parte di testing del server si sono implementati e impostati per lo sviluppo alcuni test unitari dell'applicativo. Il codice si può visionare all'interno dell'apposita cartella di test.

Particolare attenzione si è poi posta per i test di integrazione incui si è sviluppato un semplice script in fsharp che consente di simulare l'invio di dati dai valori random, strutturati come indicato nella sezione del progetto, all'endpoint specifico del server che si occupa di ricevere e salvare correttamente i dati. Questo ha consentito di parallelizzare al meglio il lavoro tra i membri del gruppo in quanto è possibile testare l'applicativo senza dover ottenere dei dati reali e conseguentemente impostare tutta l'infrastruttura ed inoltre ha impostato uno standard da seguire per l'invio dei dati verso il server che i client devono seguire affinché non si verifichino problemi. Anche questo script è presente nel repository con le relative dipendenze.