Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica (Classe L-31)

Group Project
Ethereum Thermostat

Studenti
Diomedi Diego
Caminonni Marco
Bollino Emmanuele

Supervisori
Re Barbara
Morichetta Andrea
Marcelletti Alessandro

A.A. 2019/2020

Indice

1. Introduzione

Il progetto consiste nell'implementazione di un termostato sfruttando la blockchain Ethereum. L'approccio al progetto è stato quello di partire da dei diagrammi BPMN di un termostato per arrivare ad un'applicazione mobile per gestire il tutto.

Il progetto è stato realizzato con una metodologia agile. Infatti, è stato possibile applicare cambiamenti alla specifica iniziale in corso d'opera senza troppe difficoltà. Ciò è stato reso possibile anche dal numero ridotto di persone del gruppo.

1.1 Struttura del progetto

Il progetto è strutturato nelle seguenti macro aree:

- **Diagrammi BPMN**

  Diagrammi BPMN di coreografia e collaborazione per definire il funzionamento del termostato

- **Libreria generazione codice Solidity (Solcraft)**

  Libreria per generare agevolmente codice Solidity senza ricorrere all'interpolazione di stringhe.

- **Traduttore da BPMN a Solidity**

  Traduttore per generare codice Solidity a partire da un modello di coreografia.

- **Sensori e attuatori**

  Gestione della sensoristica per la temperatura e degli attuatori per il sistema di riscaldamento.

- **Gateway**

  Gestione dei gateway collegati alla blockchain che gestiscono i sensori e gli attuatori.

- **App mobile**

  Applicazione mobile per gestire i gateway e il controllo del termostato sulla blockchain.

- **Server ausiliari**

  Creazione di server ausiliari per la compilazione dei contratti Solidity e l'utilizzo del traduttore online.

1.2 Tecnologie utilizzate

Le tecnologie utilizzate per la realizzazione dell'intero progetto sono complessivamente:

- **BPMN**
- **ChorChain**
- **Java**
- **Python**
- **Dart**
- **Flutter**
- **Web3**
- **Solidity**
- **Arduino**
- **Raspberry**
- **Bluetooth**
- **NodeJS**
- **Heroku**

L'utilizzo di queste tecnologie e i motivi delle scelte verranno discussi nei capitoli successivi.

1.3 Lavoro di gruppo

Gran parte delle tecnologie sopra elencate non facevano parte delle conoscenze dei membri del gruppo. Dunque è stata necessaria una fase di apprendimento abbastanza lunga, tale da garantire un proseguo del progetto il più lineare possibile.

Ognuno dei membri del gruppo si è focalizzato di più su alcune di queste tecnologie in modo tale da avere una buona suddivisione del lavoro.

La comunicazione all'interno del gruppo è stata molto agile e semplice. Ciò grazie al fatto del numero ridotto di persone nel team e dal continuo aggiornamento e dialogo tramite videoconferenza.

Sebbene tutti i membri abbiano partecipato ad ogni sezione in modo attivo, la divisione grossolana delle focalizzazioni di lavoro è la seguente:

- **Diomedi Diego** app mobile; comunicazione bluetooth tra app e gateway; comunicazione con la blockchain.
- **Caminonni Marco** gestione sensori e attuatori; comunicazione bluetooth tra gateway raspberry e arduino; comunicazione dei gateway con la blockchain.
- **Bollino Emmanuele** analisi dei diagrammi; libreria Solcraft; traduttore da modello di coreografia a codice Solidity.

2. Fase di avvio

La fase di avvio del progetto è durata molto rispetto alla realizzazione dell'intero progetto in quanto si è ritenuto opportuno fare un'analisi approfondita del problema da risolvere e studiare le tecnologie da utilizzare.

2.1 Diagrammi BPMN

In primis si è fatta un'analisi dei diagrammi BPMN da utilizzare come base per il termostato. Si è partiti da un modello di coreografia già fatto [figura 3.1] e lo si è fatto poi evolvere [figura 3.2].

Quindi è stato creato anche un modello di collaborazione per chiarire meglio i comportamenti interni dei componenti e i punti di comunicazione.

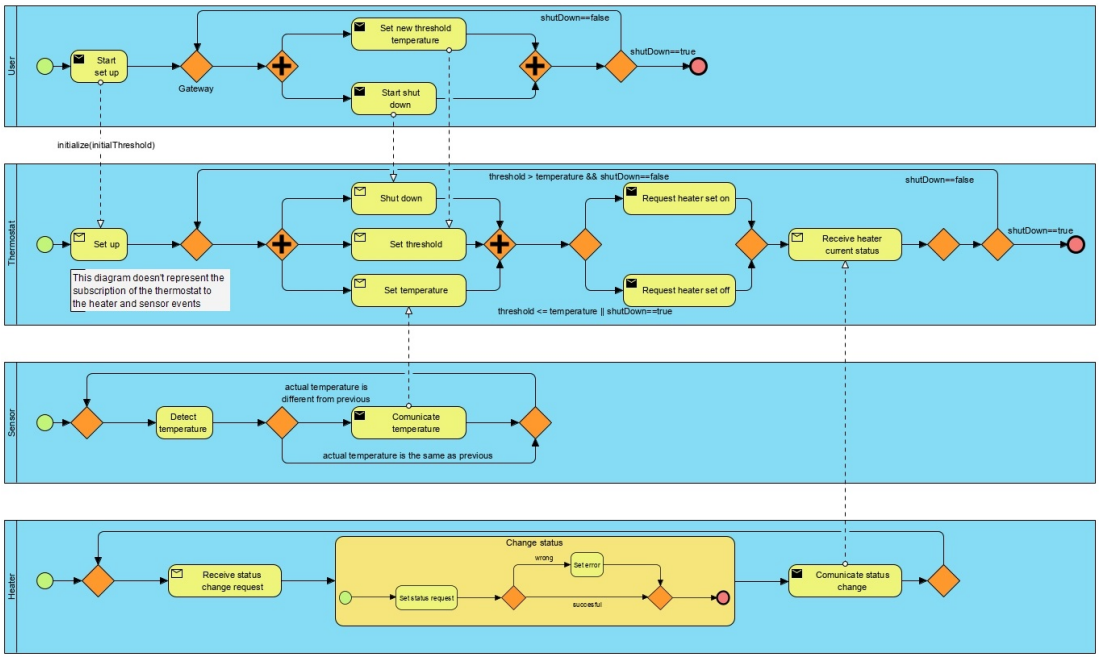

Figura 2.1: Collaboration diagram iniziale

Tuttavia poi si è continuato a sviluppare solamente il modello di coreografia che rappresenta il perno su cui ruota tutto il progetto. Infatti questo rappresenta le comunicazioni tra i vari ruoli che è ciò che serve per derivare il codice Solidity. Sebbene comunque il modello di collaborazione avrebbe potuto aiutare a definire maggiormente i comportamenti interni dei singoli ruoli. Tuttavia per lo scopo del progetto que-

sto era eccessivo in quanto il focus è sullo scambio dei messaggi tra i vari ruoli che rappresenteranno poi dei contratti Solidity o degli attori esterni.

Le evoluzioni del diagramma verranno approfondite successivamente.

2.2 ChorChain

Parte cruciale del progetto è stata l'analisi della piattaforma ChorChain per capire se questa fosse adatta allo sviluppo del termostato.

ChorChain permette infatti non solo di tradurre un modello di coreografia in un contratto Solidity ma fornisce anche una comoda interfaccia per interagire con esso. Tuttavia ChorChain genera un contratto unico che funge da mezzo di comunicazione per tutti i ruoli.

Tuttavia, dopo delle prove di scrittura dei contratti per il termostato in modo manuale, ci siamo resi conto che sarebbe stata più coerente una architettura distribuita in cui ogni ruolo rappresenta un contratto. Ad esempio un contratto Termostato a cui sono collegati i contratti Sensor e Heater che dialogano tra loro. In questo caso Sensor e Heater diventando indipendenti visto che le decisioni sull'accensione di quest'ultimo in base ai dati forniti da Sensor vengono prese da Thermostat. In questo modo si ha una migliore modularità.

Inoltre, appoggiandosi sempre a ChorChain per il dialogo tra i componenti, viene meno la logica di decentralizzazione della blockchain.

2.3 Architettura

L'architettura rappresenta l'effettiva realizzazione del termostato.

Da una prima analisi si è deciso di utilizzare Arduino per i sensori di temperatura e gli attuatori per il riscaldamento. Si è deciso poi che questi siano collegati a dei gateway Raspberry. Infine il tutto gestito tramite un'app mobile.

Maggiori dettagli sull'architettura saranno discussi in seguito.

2.4 Decisioni di realizzazione

Le decisioni prese in fase di avvio riguardano l'abbandono della piattaforma ChorChain in favore della realizzazione di un nuovo traduttore da un modello di coreografia BPMN a Solidity. La differenza sta nel fatto che il nuovo traduttore riesca a generare un'architettura a contratti multipli in cui ogni ruolo è un contratto. Maggiori dettagli sul traduttore saranno discussi in seguito.

Inoltre, un'altra decisione importante riguarda l'utilizzo massiccio della blockchain senza passare attraverso nessun server intermedio se non per delle funzioni di utilità che comunque non pregiudicano l'utilizzo decentralizzato della blockchain.

Si è poi deciso che l'applicazione mobile venisse realizzata in Dart con il framework Flutter dal momento in cui è stata verificata la disponibilità di un porting della libreria web3 per il linguaggio Dart.

2.5 Contratti iniziali

Al fine di aver ben chiaro l'obiettivo da raggiungere, inizialmente sono stati scritti a mano i contratti per il sistema termostato. In tal modo è stato possibile visualizzare quale fosse il risultato da raggiungere attraverso il traduttore automatico. Il diagramma delle classi che rappresenta i contratti inizialmente scritti è riportato in figura 2.2.

Come è possibile vedere, il meccanismo di comunicazione inizialmente pensato è basato su un publish-subcribe. Inizialmente inoltre non era prevista la possibilità di avere stanze multiple e quindi più coppie di sensore e dispositivo di riscaldamento.

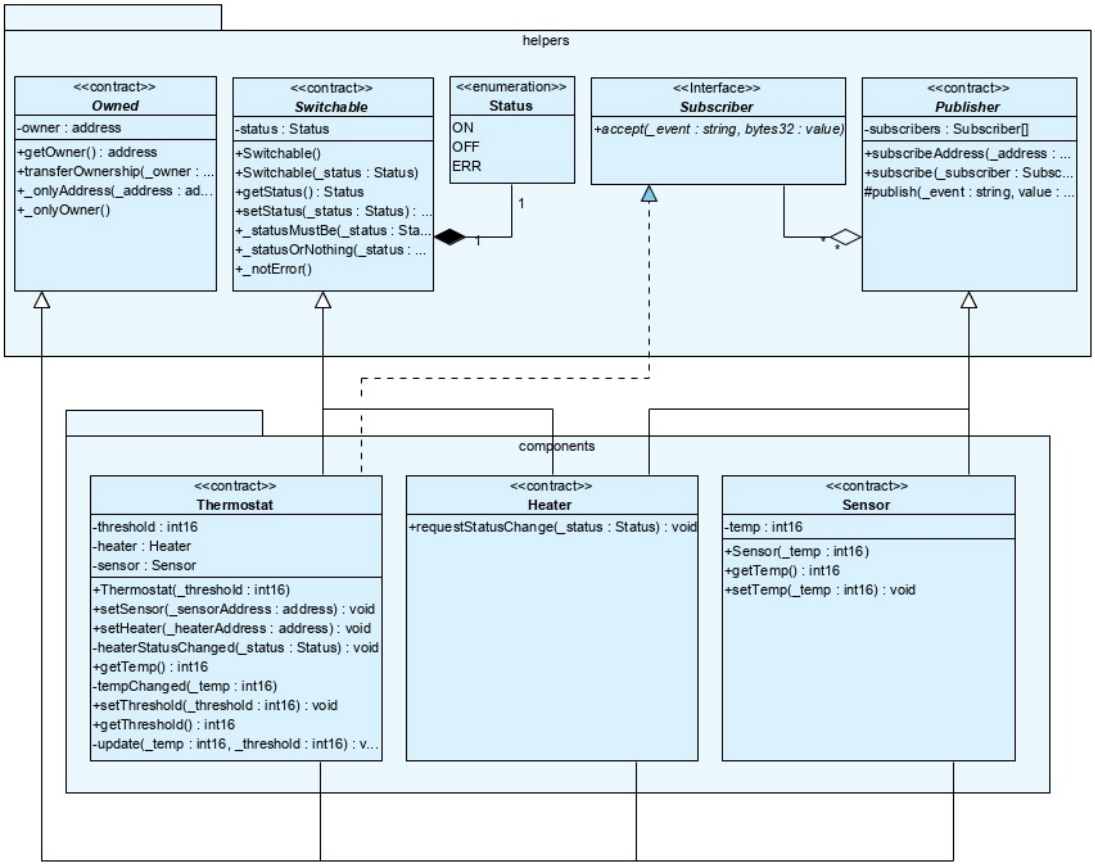

Figura 2.2: Class diagram contratti iniziali

3. Choreography diagram

3.1 Diagramma di partenza

Il diagramma di partenza è stato fornito con la traccia del progetto ed è illustrato in figura 3.1:

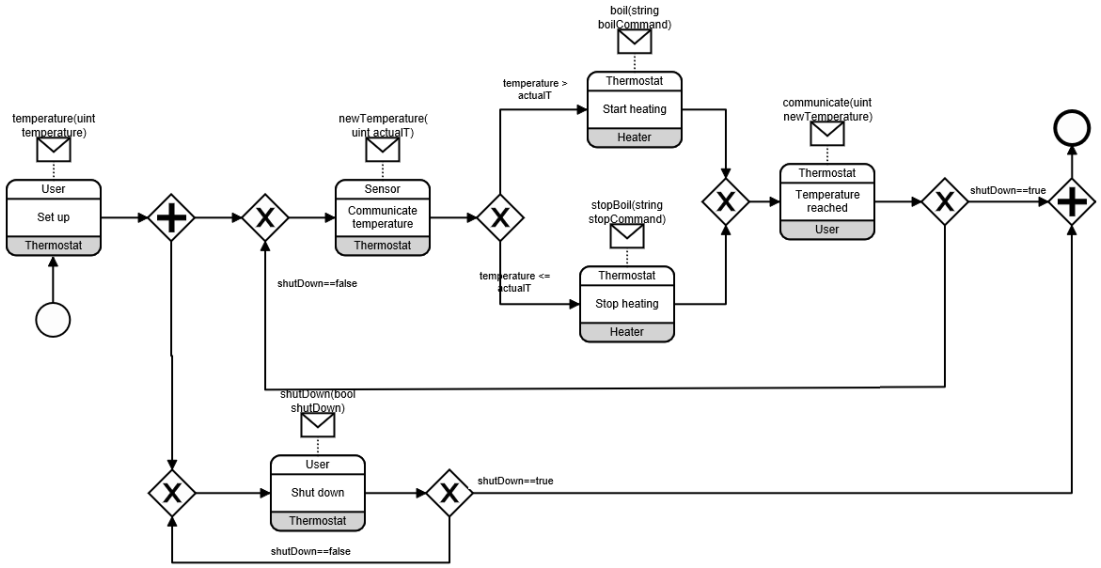

Figura 3.1: Choreography diagram iniziale

Seppur il diagramma iniziale risulti funzionante per un utilizzo basilare, è stato opportuno ridefinirlo in quanto porta con sé alcuni problemi:

Temperatura di soglia Non è possibile cambiare la temperatura di soglia dopo che è stata impostata all'inizio con il task Set Up. Questo rappresenta una grossa limitazione al funzionamento corretto di un termostato. In questo caso per cambiare la temperatura di soglia è necessario terminare il modello corrente e crearne uno nuovo.

Utenti esterni Al fine di generare un traduttore automatico, è necessario distinguere quali ruoli debbano essere dei contratti e quali no. A tal proposito, ci sono dei ruoli che vengono rappresentati da persone fisiche o comunque dispositivi che agiscono come tali. Si necessita dunque di trovare un modo per effettuare questa distinzione.

Spegnimento Non è possibile spegnere il termostato temporaneamente per poi riaccenderlo se non distruggendolo e crearne un altro. Quindi in questo caso l'azione di shut

down non spegne il termostato in vista di una possibile riaccensione, bensì lo distrugge giungendo all'end-event.

Messaggi idempotenti Il messaggio di shut down può contenere sia il valore true ché il valore false. In quest'ultimo caso, il modello si evolve senza generare cambiamenti e torna nello stato di partenza. Ciò non genera particolari problemi se non di chiarezza nell'utilizzo del sistema. Si rende necessario quindi un modo per obbligare il valore di un messaggio.

Aggiornamento stato riscaldamenti In seguito allo spegnimento/distruzione del termostato, i riscaldamenti debbono essere necessariamente spenti. In questo modello è possibile la terminazione senza che lo stato dei riscaldamenti venga aggiornato. Dunque se i riscaldamenti sono accesi al momento della distruzione, questi lo rimarranno. L'unico modo per farli spegnere è cambiare la temperatura fornita dal sensore.

Istanze multiple Qui non vengono gestite istanze multiple di sensori e riscaldamenti. Ciò significa che ci sarà un solo sensore e un solo sistema di riscaldamento a capo di un termostato. Il ché non presenta problemi e può essere semplicemente una scelta progettuale. Tuttavia si è deciso in seguito che i ruoli Sensor e Heater fossero parallel multi instance. Ciò è stato fatto perché in questo modo è possibile avere a capo di un termostato più coppie di sensori e riscaldamenti. All'atto pratico ciò si traduce ad esempio per un utilizzo domestico, nell'avere per ogni stanza un sensore e un sistema di riscaldamento autonomo rispetto alle altre stanze.

I problemi descritti hanno reso necessario evolvere il modello. Il modello finale è stato definito per evoluzioni successive nel corso dell'intero progetto. Per semplicità l'intero processo di evoluzione verrà omesso.

3.2 Diagramma finale

Il diagramma finale è illustrato in figura 3.2. Questo, come già detto, è il risultato di raffinamenti successivi perdurati per tutto il corso del progetto.

Verranno quindi discussi gli accorgimenti che sono stati adottati durante la stesura del modello di coreografia al fine di risolvere i problemi sopracitati.

3.2.1 Cambio della temperatura di soglia

La possibilità di cambiare la temperatura di soglia in ogni momento e non solo in fase di avvio, è reso possibile inserendo un loop nel diagramma e quindi il task relativo al cambio della temperatura in questo loop.

3.2.2 Istanze multiple

Al fine di consentire istanze multiple di sensori e dispositivi di riscaldamento, i relativi ruoli nel diagramma sono stati marcati come parallel multi instance. Ciò significa nel caso concreto che le coppie di sensori e dispositivi di riscaldamento possono operare parallelamente e in modo indipendente una dall'altra.

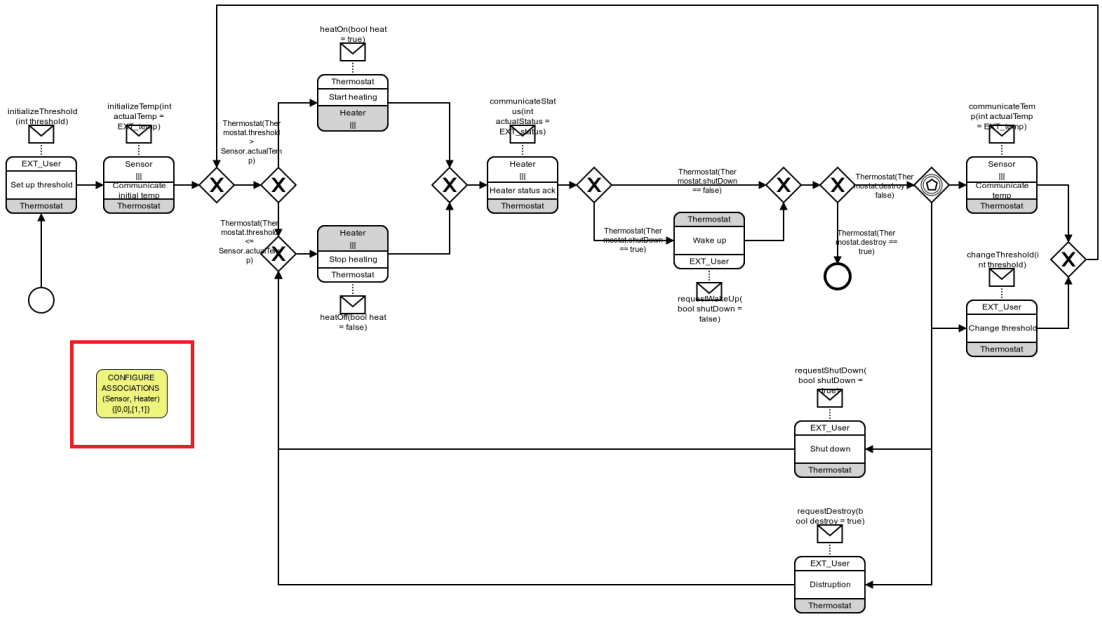

Figura 3.2: Choreography diagram finale

Associazioni

Per lo scopo del sistema termostato si è reso necessario creare degli accoppiamenti tra i ruoli multi instance. In questo caso tra sensori e dispositivi di riscaldamento in modo tale da avere un sensore e dispositivo di riscaldamento per stanza che operano in sincronia tra loro. L'associazione è stata realizzata aggiungendo un apposito task (cerchiato in rosso) di configurazione. Maggiori dettagli riguardo ai task di configurazione saranno discussi in seguito.

3.2.3 Spegnimento e distruzione

Si è reso necessario differenziare lo spegnimento del termostato con la sua distruzione. Nel primo caso il termostato può essere spento (e con lui i dispositivi di riscaldamento) per poi essere riacceso regolarmente in un futuro momento. Invece, nel caso della distruzione, tutti i contratti generati cessano di funzionare per sempre. Quest'ultimo caso è rappresentato dall'end event. Mentre invece lo spegnimento normale è realizzato attraverso un semplice task di spegnimento e il controllo del fatto che il termostato sia acceso o spento.

3.2.4 Messaggi idempotenti

L'idempotenza dei messaggi descritta precedentemente è stata risolta modificando la sintassi dei messaggi. In questo modo si vincolano alcuni messaggi ad inviare solo alcuni valori prestabiliti. Ad esempio, se il termostato è spento, non si può inviare un ulteriore messaggio di spegnimento impostando la variabile shutDown a false, bensì sarà possibile esclusivamente impostare quella variabile a true con un messaggio che forza il valore inviato. Maggiori dettagli sulla sintassi dei messaggi verrano discussi in seguito.

3.2.5 Utenti esterni

Gli utenti esterni che interagiscono con il sistema ma che non sono rappresentati dai contratti sono ruoli il cui nome ha come prefisso "EXT_". Questa scelta poteva essere rimpiazzata attraverso l'utilizzo di un attributo personalizzato all'interno del file bpmn. Tuttavia si è preferito utilizzare un prefisso nel nome per rendere più chiaro visivamente il concetto di utente esterno. Altrimenti si sarebbe dovuto realizzare un modeler specifico che fosse in grado di individuare l'attributo e visualizzarlo. In questo modo gli utenti esterni possono essere facilmente individuati e inseriti utilizzando un qualsiasi modeler. Inoltre, alcuni utenti esterni possono essere omessi come ruoli. Ciò accade nel caso in cui un ruolo che rappresenta un contratto che invia un messaggio, ha degli attributi che devono essere prima inviati dal suo proprietario esterno. In questo caso è possibile semplificare il diagramma omettendo questo task di impostazione degli attributi ma comunque utilizzando il prefisso "ext_" nei valori che il ruolo contratto invia.

3.2.6 Aggiornamento stato riscaldamenti

L'aggiornamento dei riscaldamenti ad ogni azione è stato reso possibile attraverso il loop sopracitato. Comunque sia, in seguito ad azioni di spegnimento e di distruzione, i riscaldamenti devono essere forzatamente spenti, indipendentemente dalla temperatura rilevata e dalla temperatura di soglia. Pertanto, è stato realizzato un collegamento diretto uscente da questi task ed entrante nel task di spegnimento, bypassando il controllo della temperatura.

3.2.7 Problemi rimasti

Tuttavia, questo diagramma porta con sé altri problemi trascurabili al fine del progetto ed è stato scelto volontariamente di lasciarli così per non complicare eccessivamente il diagramma e quindi i contratti generati.

Assenza di parallelismo

Il loop non prevede parallelismo per le varie azioni principali e il suo accesso è mutualmente esclusivo. In particolare, se il sensore ha appena comunicato la sua temperatura e il termostato la sta elaborando con i successivi task, l'utente non può cambiare la temperatura di soglia in quanto questo task risulterà non abilitato. Bisognerà quindi aspettare la terminazione di tutti i task che seguono.

3.2.8 Task configurazione

Al fine di utilizzare delle configurazioni nel diagramma per la traduzione, sono stati introdotti dei task di configurazione. I task di configurazione sono dei normali task che non hanno sequence flow né in entrata né in uscita e il cui nome descrive la configurazione da fare. Il nome di un task di configurazione inizia sempre con "CONFIGURE" seguito dal nome della configurazione da apportare e dagli argomenti specifici della configurazione.

Configurazione associazioni

Uno dei task di configurazione è quello per configurare le associazioni, ossia di associare i ruoli multi instance.

Si premette che ogni ruolo multi instance è identificato da un indice che va da 0 al numero massimo di istanze - 1. Questo indice lo identifica in modo univoco tra i ruoli dello stesso tipo.

Il nome della configurazione è "ASSOCIATIONS", dunque questo task avrà il nome che inizia con "CONFIGURE ASSOCIATIONS".

Gli argomenti da fornire sono strutturati in due parti. La prima è costituita dai nomi dei ruoli che si vogliono associare. Questi sono racchiusi all'interno di parentesi tonde e separati da virgole. La seconda parte è data dalle associazioni vere e proprie. Ogni associazione è un insieme di indici dei ruoli separati da virgole e racchiusi da parentesi quadre. Ogni associazione è separata da un punto e virgola e sono racchiuse tra parentesi graffe. L'ordine degli indici dei ruoli deve riflettere l'ordine dei ruoli fornito nella prima parte.

3.2.9 Sintassi e semantica dei messaggi

Ogni messaggio ha un nome e contiene un insieme di argomenti inviati ad un ruolo. Questi argomenti hanno un tipo, un nome e un valore. Il tipo riflette i tipi base del linguaggio Solidity. Il nome può essere scelto in modo arbitrario. Il valore può essere: un valore predefinito e forzato; una variabile già ottenuta in precedenza; un valore non specificato deciso al momento dell'invio del messaggio. La sintassi è identica a quella dei messaggi di ChorChain con l'aggiunta dei valori. Questi sono specificati dopo un segno di uguale dopo il nome dell'argomento.

4. Architettura

L'architettura scelta per il termostato in fase di esecuzione è in figura 4.1. L'architettura conserva principi di modularità. Sarà descritta con un approccio bottom-up.

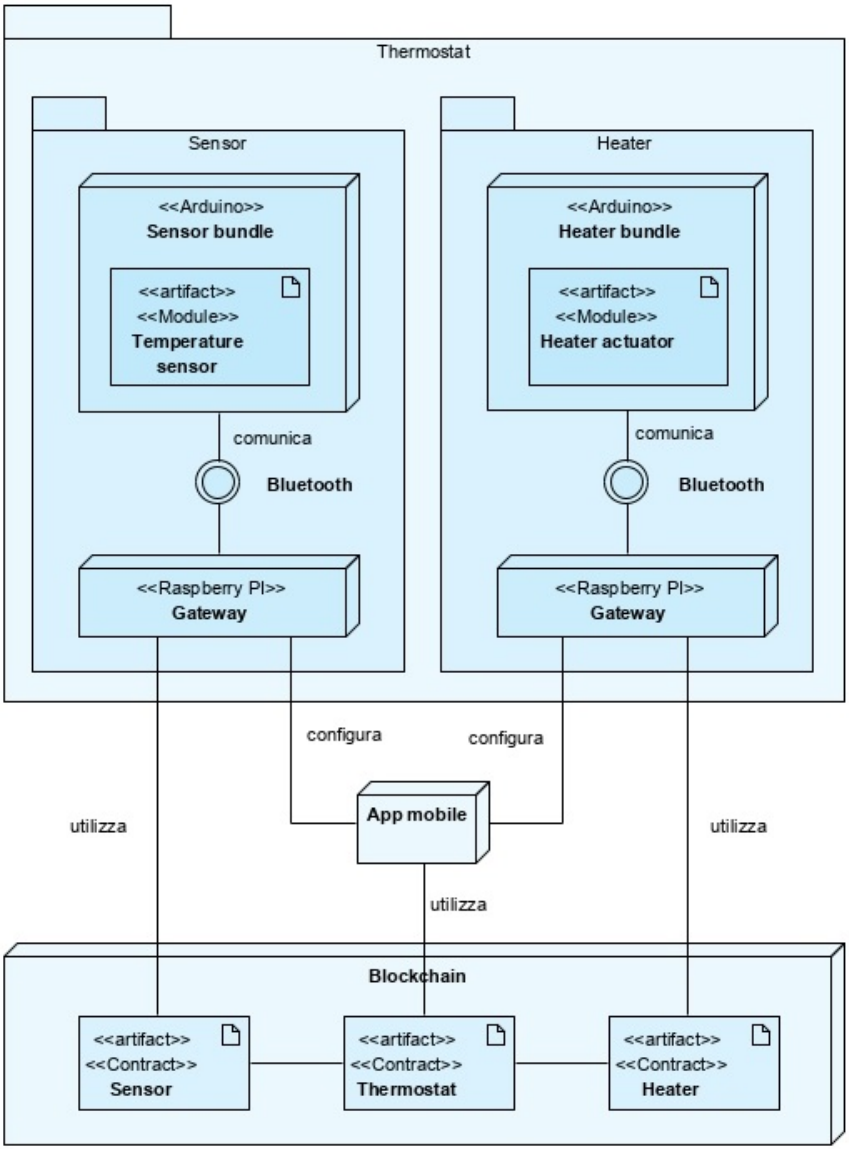

Figura 4.1: Deployment diagram con una stanza

4.1 Sensori e attuatori

4.1.1 Sensore

Il sensore di temperatura fisico è costituito da un modulo DHT11 collegato ad una scheda Arduino. L'Arduino a sua volta e equipaggiato con un modulo bluetooth HC-05. Tutto ciò costituisce il modulo fisico per il rilevamento della temperatura ambientale. Il bluetooth viene utilizzato come interfaccia di collegamento con il gateway. Ogni blocco sensore è collegato ad un unico gateway. Il blocco sensore ha quindi il compito di comunicare sporadicamente la temperatura rilevata al gateway.

4.1.2 Dispositivo di riscaldamento

L'attuatore del sistema di riscaldamento fisico è costituito da un relay che accede o spegne un modulo di riscaldamento ignoto. Infatti questo non è stato realmente implementato nel progetto. Infatti, la scelta del relay è scaturita dal momento in cui il dispositivo reale di riscaldamento è ignoto e quindi utilizzando un relay si rende il tutto modulare e facilmente configurabile. Per semplicità, al momento è stato utilizzato un semplice LED che indica se sta riscaldando o meno. Il relay è gestito da una scheda Arduino, equipaggiata con un modulo bluetooth HC-05. Il tutto costituisce il blocco riscaldamento. Il bluetooth serve come interfaccia di comunicazione con il relativo gateway. Ogni blocco riscaldamento è collegato ad un unico gateway. Il compito del modulo riscaldamento è quindi quello di ricevere dal proprio gateway il comando di accensione/spegnimento e di ritornare un feedback riguardo l'effettiva accensione/-spegnimento. Questo perché non è detto che il modulo di riscaldamento riesca ad adempiere ai comandi del gateway: potrebbero accadere impedimenti fisici.

4.1.3 Gateway

I gateway sopracitati sono costituiti da dei Raspberry PI 3, i quali sono già dotati di modulo bluetooth. I gateway fungono da ponte tra i sensori/attuatori e la blockchain. Questi, infatti, si occupano di effettuare i deploy dei contratti, effettuare transazioni, ascoltare eventi, leggere valori nella blockchain e comunicare/leggere valori verso/da i blocchi sensore/riscaldamento.

4.1.4 Blockchain

I contratti in blockchain sono indipendenti e comunicanti. In particolare il contratto Thermostat si occupa di gestire la logica di accensione dei riscaldamenti. Il contratto Sensor si occupa di memorizzare la temperatura rilevata e di comunicarla al termostato. Il contratto Heater si occupa di ricevere il comando di accensione da parte del termostato e di restituirne il feedback di stato. I contratti Sensor e Heater sono quindi completamente indipendeti tra loro in quanto il contratto Thermostat funge da intermediario. La logica associativa è contenuta all'interno del contratto Thermostat in quanto è esso che gestisce le coppie di Sensor e Heater.

4.1.5 App mobile

L'app mobile è il fulcro dell'utilizzo del sistema termostato. Questa infatti esegue il deploy del contratto Thermostat e configura i gateway relativi al sensore e dispositivo

di riscaldamento. Infatti, attraverso di questa è possibile scegliere inizialmente quante stanze il termostato debba avere; si possono configurare i gateway indicando a quali blocchi sensore/riscaldamento debbano essere collegati. L'utente finale attraverso l'app mobile cambia la temperatura di soglia, spegne e accende il termostato, distrugge l'intero sistema, ne crea uno nuovo e legge la temperatura attuale. L'app mobile inoltre utilizza un server che fornisce bytecode a ABI del contratto Thermostat in base al numero di stanze desiderate.

4.1.6 Stanze multiple

La questione di avere stanze multiple con sensore e dispositivo di riscaldamento associati viene realizzata attraverso app mobile con la possibilità di visualizzare la temperatura e lo stato del sistema di riscaldamento in ogni stanza. Poi con la possibilità di duplicare i blocchi sensore e blocchi riscaldamento. I gateway sensore e gateway riscaldamento potranno quindi essere collegati rispettivamente a più blocchi sensore e blocchi riscaldamento. All'aumentare dei blocchi sensore/riscaldamento, aumentano anche i contratti deployati nella blockchain. Infatti, ad ogni sensore e dispositivo di riscaldamento corrisponde un contratto. Nel contratto Thermostat sarà presente quindi un'ulteriore associazione. Il numero di stanze deve essre deciso in partenza in quanto il codice Solidity dei contratti è variabile in funzione di queste. Il diagramma di deploy con più stanze è riportato in figura 4.2.

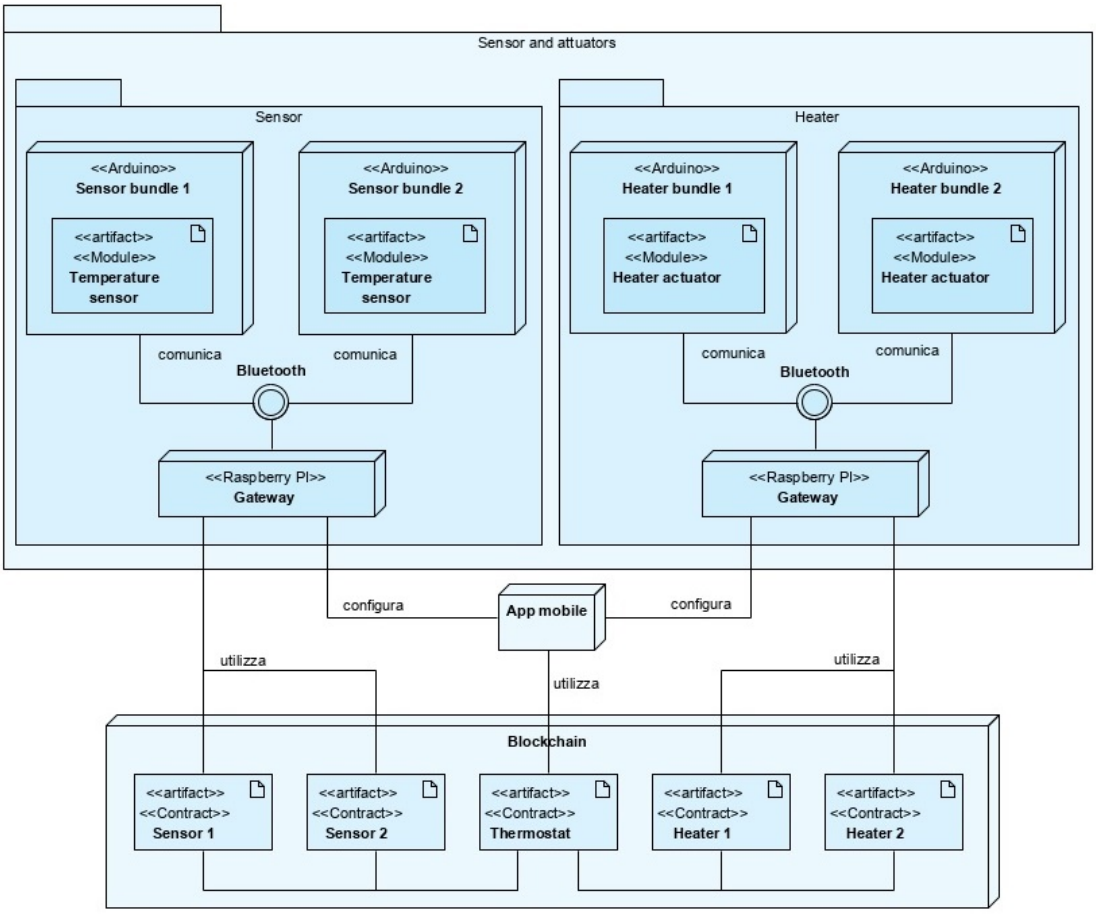

Figura 4.2: Deployment diagram con due stanze

5. Solcraft

Solcraft è una libreria Java che permette la creazione di codice Solidity in maniera semplice, veloce, strutturata e precisa. La libreria riflette il modo di scrittura del programmatore e quindi è di immediata comprensione.

Si basa sulla grammatica ufficiale del linguaggio Solidiy. La libreria è stata creata al fine di supportare il traduttore da un modello di coreografia ad un codice Solidity. Pertanto, non sono stata implementata tutta la grammatica ma la gran parte che serve per scrivere dei contratti Solidity di base.

Solcraft formatta direttamente il codice in maniera elegante. Il codice viene scritto componendo oggetti di classi che rappresentano i componenti Solidity. Aggregando insieme queste classi, è possibile scrivere un intero file Solidity.

5.1 Struttura classi

L'elenco completo delle classi costituenti la libreria è possibile trovarlo all'interno della libreria stessa ed è superfluo riportarlo qui. Si riporta in figura 5.1 un diagramma delle classi contenente solo una piccola porzione delle classi totali utilizzabili in Solcraft. Solamente il package soliditycomponents che serve per creare codice Solidity contiene 33 classi allo stato attuale. Sono anche presenti package che contengono delle classi helper (ed esempio una classe che divide un file Solidity in più file, uno per contratto/interfaccia) e componenti standardizzati (ad esempio l'Owned contract che rappresenta un contratto da estendere che ha un proprietario).

Il meccanismo della composizione e dell'ereditarietà sono fondamentali. Le collaborazioni tra le varie classi sono molteplici.

Il cuore centrale della stampa è il metodo print contenuto nell'interfaccia SolidityComponent. Questo permette di stampare in maniera formatta il codice Solidity del componente. Ogni componente implementerà quindi le sue regole. Inoltre, il metodo di default printWithIndentation stampa il componente secondo il livello di indentazione fornito.

5.2 Pubblicazione su Maven Central

La libreria Solcraft è stata pubblicata nel repository Maven Central. Ciò permette alla libreria di essere aggiunta da chiunque in qualsiasi progetto Java in modo semplice se si utilizza un gestore di progetti come Maven [Codice 5.1] o Gradle [Codice 5.2].

La pubblicazione è stata subordinata ad un'approvazione da parte del team di Open Source Software Repository Hosting (OSSRH). Dunque il progetto è stato configurato con le direttive maven per il build nel repository OSSRH dedicato.

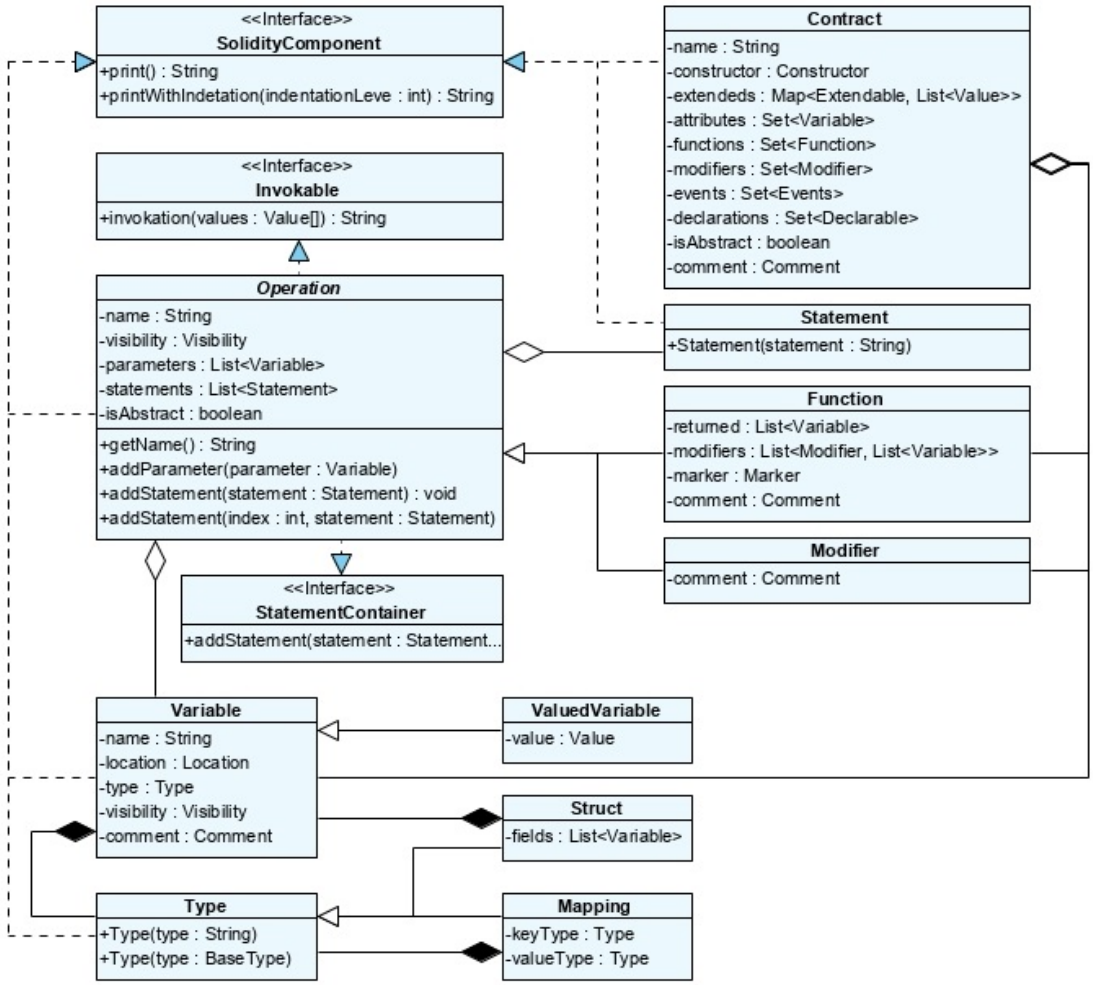

Figura 5.1: Class diagram ridotto di Solcraft

5.2.1 Info

Source code github.com/EmmanueleBollino/solcraft

Maven Central page search.maven.org/artifact/com.github.EmmanueleBollino/solcraft

```
<dependency>
  <groupId>com.github.EmmanueleBollino</groupId>
  <artifactId>solcraft</artifactId>
  <version>2.0.2</version>
</dependency>
```

Codice 5.1: Dipendenza Maven per Solcraft

```
implementation 'com.github.EmmanueleBollino:solcraft:2.0.2'
```

Codice 5.2: Dipendenza Gradle Groovy per Solcraft

6. Traduttore da BPMN a Solidity

Il traduttore è un applicativo Java che prende in input un modello di coreografia BPMN e restituisce la relativa rappresentazione a contratti multipli attraverso codice Solidity che potrà poi essere compilato. L'algoritmo crea all'inizio un file Solidity e analizzando i vari componenti del modello aggiunge poi contratti, funzioni, attributi, modificatori, ecc.

Le librerie utilizzate sono: Solcraft per la scrittura dei contratti e Camunda BPMN Model per la lettura del modello.

6.1 Algoritmo di traduzione

L'algoritmo di traduzione è abbastanza complesso perché richiede molteplici controlli sui task nel diagramma. L'algoritmo è il risultato della risoluzione alle domande su come dovessero essere ottenuti i contratti risultanti a partire da una lettura del modello.

Quali sono i contratti? Dal momento in cui l'architettura scelta è a contratti multipli, ci si è posti l'interrogativo su come capire quali dovessero essere i contratti. Ogni ruolo è un contratto ad eccezione dei ruoli esterni che sono marcati con il prefisso "EXT_ (come descritto in precedenza). Dunque, per ogni ruolo non esterno viene creato un contratto il cui nome è il medesimo.

Come gestire la proprietà? Si necessita gestire la proprietà di un contratto, ovvero assicurarsi che certe operazioni vengano effettuate solo dal proprietario. Per far ciò si è deciso di creare un contratto astratto predefinito "Owned" che viene esteso da tutti gli altri contratti. Il contratto Owned imposta come proprietario colui che effettua la transazione di deploy del contratto e contiene funzioni e modificatori utili alla gestione della proprietà, come ad esempio il modificatore "onlyOwner" assicura che una funzione venga chiamata solamente dal proprietario.

```
abstract contract Owned {
    address private owner;

    constructor() public {
        owner = msg.sender;
    }

    /**
     * Restricts access to the provided address.
     */
```

```
    modifier onlyAddress(address _address) {
        require(msg.sender == _address, "Address not allowed");
        _;
    }

    /**
     * Restricts access to the modified function only to the owner.
     */
    modifier onlyOwner() {
        require(msg.sender == owner, "Address not valid");
        _;
    }

    /**
     * Transfers the ownership of the contract.
     */
    function transferOwnership(address _newOwner) external onlyOwner {
        require(validateAddress(_newOwner), "Address not valid");
        owner = _newOwner;
    }

    /**
     * Checks whether an address is valid or not.
     * An invalid address means a full zero address.
     */
    function validateAddress(address _address)
        internal
        pure
        returns (bool isValid)
    {
        return _address != address(0);
    }
}
```

Codice 6.1: Contratto Owned

Quali sono le funzioni? Decidere quali sono le funzioni da inserire all'interno dei contratti al fine di consentire lo scambio dei messaggi è stata una scelta fondamentale. Si è scelto un approccio concettualmente semplice: per ogni messaggio da un ruolo A ad un ruolo B, il ruolo A contiene una funzione che invia i valori al ruolo B, mentre il ruolo B contiene una funzione setter che imposta i valori ricevuti. Ad esempio il messaggio in figura 6.1 è stato tradotto nel codice 6.2.

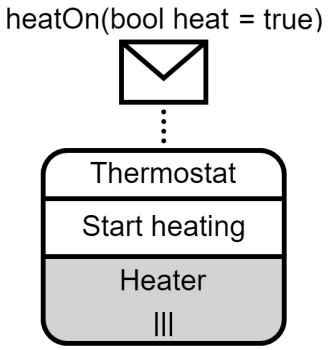

Figura 6.1: Heat on task

```
contract Thermostat {
    ...
    /**
     * Sender function from this contract to Heater
     * Associated task name: Start heating
     * Associated task id and hash: ChoreographyTask_1hyjjdu | -327002890
     * Associated source message: heatOn(bool heat = true)
     * Participants: Thermostat -> Heater
     */
    function send_heatOn() public {
        // TODO check sender (not if is internal)
        bool _enabled = false;
        for (uint256 i = 0; i < associations.length; i++)
            if (isEnabled(-327002890, associations[i])) {
                if (!_enabled) _enabled = true;
                disable(-327002890, associations[i]);
                getHeater(associations[i]).heatOn(true);
            }
        require(_enabled, "Not enabled");
    }
    ...
}

contract Heater {
    ...
    /**
     * Setter function for values provided by a contract
     * Associated task name: Start heating
     * Associated task id and hash: ChoreographyTask_1hyjjdu | -327002890
     * Associated source message: heatOn(bool heat = true)
     * Participants: Thermostat -> Heater
     */
    function heatOn(bool _heat) public onlyAddress(address(thermostat)) {
        // set all received values
        heat = _heat;
        // emit all changed values events
        emit heatChanged(heat);
        // enable next
        enable(128562386);
    }
    ...
}
```

Codice 6.2: Traduzione task heat on

Quali sono gli attributi? Gli attributi di ogni contratto sono gli argomenti che vengono ricevuti nei messaggi. Quindi per ogni messaggio ricevuto non c'è solo una setter function ma i valori settati da tale funzione rappresentano gli attributi del contratto. Oltre a questi attributi vengono aggiunti i riferimenti degli indirizzi di ogni contratto con cui si comunica e di ogni utente esterno che interagisce con il ruolo. Infine, vengono aggiunte le strutture dati necessarie per memorizzare l'avanzamento nel moello. Un esempio del contratto Heater è riportato nel codice 6.3.

Quali sono gli eventi? Per ogni attributo che viene cambiato viene generato un evento. Tutti gli attributi sono quindi pubblici e possono essere letti sia direttamente ché tramite i relativi eventi generati. Inoltre viene generato un evento per ogni pro-

gressione nel diagramma, ossia per ogni nuova attivazione/disattivazione dei task. Un esempio del contratto Heater è riportato nel codice 6.3.

```
contract Heater {
    Thermostat public thermostat;
    bool public isStarted = false;      // when starts
    bool public isEnded = false;        // when reaches the end event
    mapping(int256 => bool) public activations;
    bool public heat;
    int256 public status;

    event isStartedEvent();             // when this contract starts
    event enabled(int256 _enabledId);   // when a task is enabled
    event disabled(int256 _disabledId);
    event heatChanged(bool _heat);
    event statusChanged(int256 _status);
}
```

Codice 6.3: Attributi ed eventi del contratto Heater

Quali sono i modificatori? Non ci sono particolari modificatori da evidenziare se non l'utilizzo di quelli nel contratto Owned che regolano l'accesso alle funzioni e il modificatore che controlla che il contratto sia attivo.

Come comunicano i contratti? I contratti comunicano tra loro attraverso le funzioni setter. Inoltre ci sono funzioni speciali che notificano altri contratti dell'attivazione di alcuni task. Il contratto ricevente dovrà quindi abilitare la ricezione di certi messaggi oppure direttamente inviarne alcuni.

Come assicurare la sequenzialità? La riproduzione della sequenzialità del diagramma nei contratti è realizzata attraverso degli attributi di abilitazione [Codice 6.4]. Ogni task ha un codice identificativo e queste strutture dati segnano quali task sono abilitati e quali no. I vari contratti possono comunicare tra loro per segnalare uno con l'altro l'attivazione/disattivazione di determinati task.

```
mapping(int256 => bool) public activations;
```

Codice 6.4: Enabling mapping

Come gestire i gateway? La gestione dei gateway in una architettura a contratti multipli è assai complessa ma si è cercato di semplificarla il più possibile. Un contratto può essere interessato o meno all'attivazione di un determinato gateway. Ciò significa che un contratto che non è compreso nei task successivi all'apertura di un gateway, non è interessato all'attivazione di questo. Poi, si è scelto di non considerare affatto i gateway chiusi (con più flussi in input e uno solo di uscita), come se non fossero presenti nel modello.

Conditional gateway Per i gateway condizionali c'è una sintassi specifica per le condizioni che indica quale contratto debba verificare la condizione e su quali attributi (es: `Thermostat(Thermostat.threshold > Sensor.actualTemp)`). Nel contratto che deve controllare la condizione, verrà creata una apposita funzione che in seguito al controllo prenderà delle decisioni di flusso [Codice 6.5].

```
/**
* GATEWAY FUNCTION
* Gateway hash id: -1610351334
* Gateway id: ExclusiveGateway_1989mhw
*/
function exclusiveGateway_1989mhw() public {
    bool _enabled_send_heatOn = false;
    bool _enabled_send_heatOff = false;
    bool _enabled = false;
    for (uint256 i = 0; i < associations.length; i++)
        if (isEnabled(-1610351334, associations[i])) {
            if (!_enabled) _enabled = true;
            disable(-1610351334, associations[i]);
            if (
                threshold <=
                sensorValues[associations[i].sensorIndex].actualTemp
            ) {
                enable(-307602952, associations[i]);
                _enabled_send_heatOff = true;
            }
            if (
                threshold >
                sensorValues[associations[i].sensorIndex].actualTemp
            ) {
                enable(-327002890, associations[i]);
                _enabled_send_heatOn = true;
            }
        }
    require(_enabled, "Not enabled");
    if (_enabled_send_heatOff) send_heatOff();
    if (_enabled_send_heatOn) send_heatOn();
}
```

Codice 6.5: Conditional gateway function

Parallel gateway I parallel gateway sono gestiti semplicemente con delle funzioni che attivano tutti i task dopo il gateway parallelo. Ciò garantisce il proseguimento parallelo dei task.

Event based gateway I gateway di tipo event based sono gestiti come i parallel gateway con l'aggiunta del fatto che all'attivazione di uno dei task successivi al gateway, vengono disattivati tutti i task precedentemente attivati [Codice 6.6], notificano gli altri contratti se necessario.

```
/**
* EVENT BASED GATEWAY DEACTIVATION FUNCTION.
* This function should be called when a task after an event based gateway is
    called.
* This function deactivates all the other tasks of this contract after the
   gateway.
* Gateway id: EventBasedGateway_09o8x0n.
*/
function entered_eventBasedGateway_09o8x0n() public {
    for (uint256 i = 0; i < associations.length; i++) {
        disable(-937354887, associations[i]);
        disable(-1113449236, associations[i]);
        disable(-877216409, associations[i]);
        disable(-947496040, associations[i]);
```

```
    }
}
```

Codice 6.6: Entered event based function

Come gestire le istanze multiple? La gestione delle istanze multiple è assai complessa in quanto i contratti che comunicano con altri contratti multipli devono poter gestire le stesse funzionalità ma con contratti diversi distinguendo le varie istanze. Per scelta progettuale il numero di istanze è deciso a compile-time hard coded nel modello bpmn. Per ogni partecipante è presente l'attributo max-instances. La gestione delle istanze multiple è esclusivamente parallela, non si possono avere istanze multiple sequenziali. Ogni contratto che dialoga con un contratto multi-instance avrà una struttura dati contenente i valori ricevuti da tale contratto e un vettore di questi valori [Codice 6.7]. Nella struttura è contenuto anche l'indirizzo del contratto. Poi c'è un mapping che mappa un indirizzo con il relativo indice nel vettore dei valori. La struttura dati è ridondante ma consente l'accesso diretto per molte operazioni risparmiando poi in computazione.

```
struct SensorValues {
    Sensor sensor;
    uint256 associationIndex;
    int256 actualTemp;
}

struct HeaterValues {
    Heater heater;
    uint256 associationIndex;
    int256 actualStatus;
}

SensorValues[2] public sensorValues;
mapping(Sensor => uint256) public sensorIndex;
HeaterValues[2] public heaterValues;
mapping(Heater => uint256) public heaterIndex;
```

Codice 6.7: Gestione istanze multiple in Thermostat

Come gestire le associazioni? Le associazioni tra i contratti ad istanza multipla servono per garantire una corretta comunicazione e svolgimento del modello. Si può affermare che per ogni associazione esiste un unico stato del diagramma e viceversa. Quindi ci sono più modelli in esecuzione parallelamente. Questo parallelismo è gestito attraverso un'apposita struttura dati [Codice 6.8] inizializzata nel costruttore [Codice 6.9] di tutti i contratti che dialogano con contratti multi instance. La configurazione delle associazioni è data dal task CONFIGURE ASSOCIATIONS [Figura **??**].

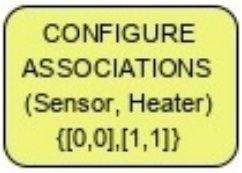

Figura 6.2: Configure Associations task

```
struct Association {
    uint256 sensorIndex; // index of the participant Sensor in the
        association
    uint256 heaterIndex; // index of the participant Heater in the
        association
    uint256 id; // id of the association, it isn't always the index of the
        associations array
    mapping(int256 => bool) activations; // activation states related to the
         association
}

Association[2] public associations;
```

Codice 6.8: Gestione associazioni in Thermostat

```
constructor() public {
    associations[0] = Association(0, 0, 0);
    sensorValues[0].associationIndex = 0;
    heaterValues[0].associationIndex = 0;

    associations[1] = Association(1, 1, 1);
    sensorValues[1].associationIndex = 1;
    heaterValues[1].associationIndex = 1;
    ...
}
```

Codice 6.9: Inizializzazione associazioni in Thermostat

In conclusione, l'algoritmo di traduzione esamina il diagramma task dopo task e verificando la tipologia dei partecipanti e del messaggio genera la giusta funzione da inserire all'interno dei contratti. Lo stesso con i gateway, esaminando i task precedenti e successivi.

7. Server accessori

Al fine di integrare le varie componenti del progetto è stato necessario realizzare dei server accessori che mettono a disposizione determinate funzionalità on demand.

7.1 Compilatore Solidity

Il server di compilazione Solidity è stato necessario in quanto all'interno del progetto serviva compilare il contratto Thermostat all'interno dell'app mobile. Questa però, essendo scritta in Dart e inserita su un sitema operativo Android, non dispone della possibilità di richiamare un compilatore Solidity. Si è scelto quindi di creare un server ad-hoc che svolgesse questa unica funzione.

Il server è scritto in Javscript con NodeJS ed Express. Questa scelta è stata fatta perché su NPM è disponibile il pacchetto solc. In realtà, è l'unico pacchetto solc che esiste integrato in un linguaggio di programmazione, infatti l'alternativa costretta sarebbe stata di installare il compilatore direttamente in un VPS e poi creare un servizio web che andasse a richiamare tramite shell il compilatore. Questa ultima scelta ci è sembrata meno pulita e più laboriosa.

Il codice di questo server è corto e di immediata comprensione [Codice 7.1] in quanto non fa altro che ricevere una chiamata POST il cui body contiene un JSON [Codice 7.2] che verrà dato in input al compilatore e il risultato sarà direttamente il risultato della chiamata che conterrà: ABI, Bytecode, eventuali errori e warning e tutte le altre informazioni necessarie.

```
import express from 'express';
import solc from 'solc';

const app = express();
const port = 80;

app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.get('/translate', (req, res) => {
  var input = req.body;

  var output = JSON.parse(solc.compile(JSON.stringify(input)));

  res.send(output);
});

app.listen(port, () => {
  console.log(`Listening at http://localhost:${port}`)
});
```

Codice 7.1: Server di compilazione Solidity

```
{
   "language":"Solidity",
   "sources":{
      "test.sol":{
         "content":"contract A { function f() public { } }"
      },
      "test2.sol":{
         "content":"import \"test.sol\"; contract B { A public a; function f
             () public { } }"
      }
   },
   "settings":{
      "outputSelection":{
         "*":{
            "*":[
               "*"
            ]
         }
      }
   }
```

Codice 7.2: Input compilatore Solidity

Questo server è disponibile all'indirizzo sol-compiler.herokuapp.com. Il deploy è stato effettuato con Heroku.

7.2 Traduttore

Il server di traduzione è necessario per avere il traduttore sempre disponibile per qualsiasi tipo di contratto, è il naturale proseguo del software di traduzione. Con una semplice chiamata è possibile caricare il modello BPMN come file e ricevere i contratti tradotti [Codice 7.3].

Inoltre, questo server serve al progetto termostato per fornire il codice da compilare. Mette a disposizione delle semplici chiamate che traducono il modello del termostato in una forma opportuna in base al numero di stanze scelte. Per rendere ancora più semplice lo sviluppo dell'app mobile, questo server mette a disposizione una chiamata che dato in input il numero di stanze, restituisce ABI e Bytecode di tutti i contratti tradotti e compilati [Codice 7.4].

```
curl --location --request GET 'http://bpmn2sol-server.herokuapp.com/
    translator/choreography' \
--form 'model=@/C:/Users/emman/Desktop/thermostat.bpmn' \
--form 'splitContracts=true'
```

Codice 7.3: Chiamata CURL traduttore

```
GET /predefined/thermostat/compiled?rooms=2 HTTP/1.1
Host: bpmn2sol-server.herokuapp.com
```

Codice 7.4: Chiamata HTTP contratto termostato con stanze

Il server è stato realizzato in Java con Spring Boot e Spring Web così da scrivere velocemente delle API REST.

Questo server è disponibile all'indirizzo bpmn2sol-server.herokuapp.com. Il deploy è stato effettuato con Heroku.

8. Sensori e attuatori

8.1 Gateway

I gateway rappresentano il ponte tra la blockchain e i blocchi sensore/heater. Questi si occupano di gestire più sensori/heater (in base al numero di stanze) dialogando con i relativi contratti.

I gateway sono implementati tramite delle schede Raspberry PI 3. Gli script sono scritti in Python. Questa scelta è stata presa in seguito alla scarsa reperibilità di librerie bluetooth aggiornate in Java. Si è deciso quindi di cambiare linguaggio e proseguire con Python. Questa scelta si è rivelata ottimale in quanto la comunicazione con il bluetooth è estremamente più semplice e immediata, così come l'utilizzo delle librerie web3. Dunque, come libreria per la comunicazione bluetooth è stata scelta Pybluez, mentre per la comunicazione con la blockchain Web3.py. La comunicazione con il bluetooth avviene utilizzando il protocollo RFCOMM.

Compiti I gateway assolvono il compito di interfacciare i moduli fisici con i contratti nella blockchain. I Gateway utilizzati sono due: uno per i sensori e uno per i dispositivi di riscaldamento. Indipendentemente dal numero di stanze, il numero di gateway non cambia. Ogni gateway infatti può gestire un numero illimitato di sensori/dispositivi di riscaldamento. Ad ogni sensore/dispositivo di riscaldamento fisico corrisponde un relativo contratto nella blockchain. Il gateway si occupa di associarli e gestire la comunicazione in ambo i sensi.

Configurazione La configurazione iniziale dei gateway riguarda l'associazione tra i gateway stessi e i sensori/dispositivi di riscaldamento fisici. Questo viene fatto attraverso l'app mobile. I passi di configurazione sono i seguenti:

1. L'app mobile crea un contratto Thermostat in base al numero di stanze desiderato
2. L'app mobile esegue il deploy del contratto Thermostat
3. L'app mobile scansiona i dispositivi bluetooth circostanti e riconosce quali possono essere dei gateway.
4. L'utente sceglie quali sono i gateway desiderati e l'app li memorizza.
5. Ciascun gateway esegue la scansione dei dispositivi bluetooth nelle vicinanze e riconosce quali sono i moduli fisici con cui può interfacciarsi.
6. Ciascun gateway restituisce all'app la lista dei dispositivi a cui può collegarsi.
7. L'utente sceglie quali sono i dispositivi in base al numero di stanze.

8. L'app comunica ai gateway quali sono i dispositivi scelti.
9. Ciascun gateway esegue il deploy dei relativi contratti e memorizza con sé l'associazione tra modulo fisico e il relativo contratto.
10. Le varie parti iniziano a cooperare esclusivamente tramite blockchain.

Comunicazioni multiple Come già detto, un gateway può gestire più moduli. Le comunicazioni avvengono tramite bluetooth e il problema è quello di collegare il bluetooth del gateway a più moduli. L'idea iniziale che è stata poi abbandonata è stata quella di creare una rete token-ring. In questo modo però tutti i moduli avrebbero dovuto comunicare tra loro e il moduli non sarebbero più stati indipendenti. Quindi si è deciso di mantere la centralità dei gateway con una topologia a stella. Tuttavia, rimangono i problemi di accesso. La soluzione scelta è una sorta di polling sequenziale: il gateway si connette ad un modulo per volta e nel caso dei sensori, il gateway decide quando connettersi a quale sensore e quindi chiedere la temperatura, mentre nel caso degli heater il gateway si connette ad un heater solo quando deve impartirgli un comando e rimane connesso a questo finché non riceve la conferma di accensione/spegnimento.

8.2 Moduli

Come già citato in precedenza, i moduli sono costituiti da delle schede Arduino equipaggiate con dei moduli bluetooth HC-05 che consentono la comunicazione con i gateway.

Sensori I moduli sensore dispongono di un sensore DHT per il rilevamento della temperatura. Questo sensore è in grado di rilevare anche il livello di umidità ambientale ma questa funzionalità non viene utilizzata. Il modulo sensore emette la temperatura attuale ad intervalli regolari ai dispositivi collegati al bluetooth. Lo schema è rappresentato in figura 8.1.

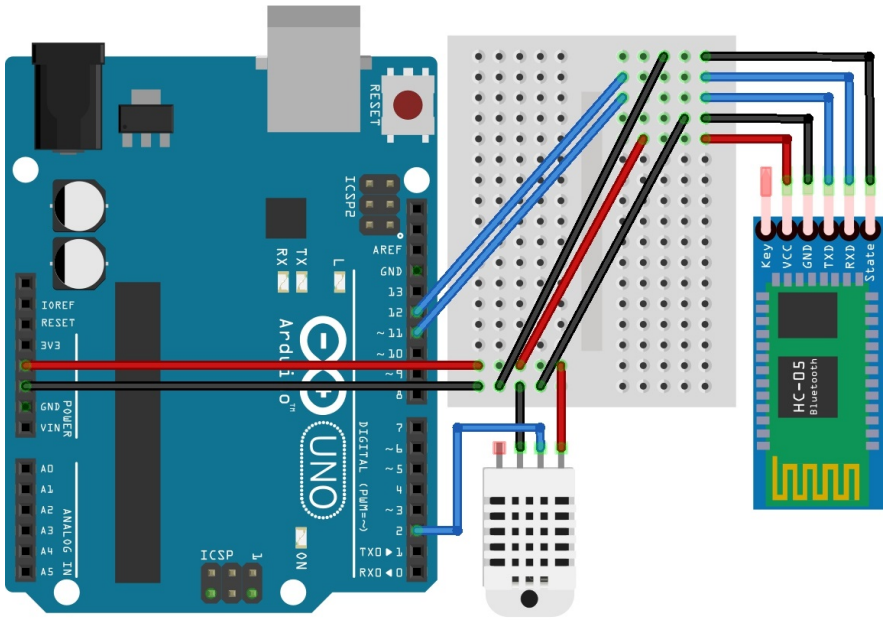

Figura 8.1: Schema collegamento modulo sensore

Heater I moduli riscaldamento dispongono di un relay al quale può essere collegato qualsiasi tipo di sistema di riscaldamento personalizzato. Tuttavia, per semplicità, in questo progetto è stato utilizzato un semplice diodo LED alimentato dalla scheda Arduino. Lo sketch del programma Arduino consiste nel ricevere un comando di accensione/spegnimento dal dispositivo bluetooth collegato ed aprire o chiudere il relay. Per sistemi di riscaldamento più complessi, in seguito al comando ricevuto, il modulo può restituire un eventuale errore e/o lo stato attuale del sistema di riscaldamento. Lo schema è rappresentato in figura 8.2

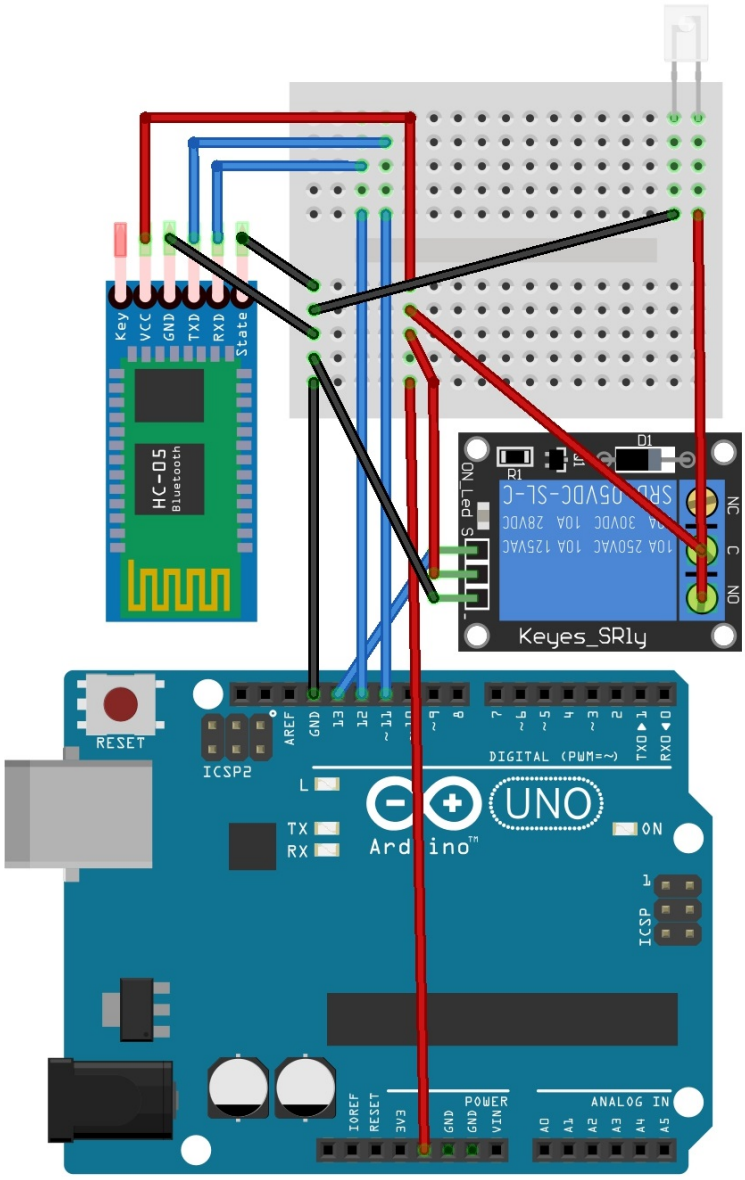

Figura 8.2: Schema collegamento modulo riscaldamento

9. App mobile

L'app mobile è il cuore dell'esperienza utente del sistema termostato. Questa applicazione consente di eseguire tutto e nasconde all'utente i dettagli tecnici per una UX avanzata. L'app è scritta in Dart utilizzando il framework Flutter. Per la comunicazione con la blockchain è stata utilizzata la libreria web3dart.

9.1 Funzionalità

L'applicazione dispone delle seguenti funzionalità:

Caricare un wallet L'applicazione per funzionare necessita di un wallet collegato con un plafond non vuoto. Ciò perché la maggior parte delle transazioni sono a pagamento. L'app consente di configurare un wallet tramite il semplice inserimento della chiave privata. L'utente potrà quindi creare il suo wallet tramite applicazioni di terze parti più conosciute come ad esempio Metamask. L'utente potrà avere sempre sotto controllo il costo del termostato tramite il saldo disponibile nel proprio wallet [Figura 9.1].

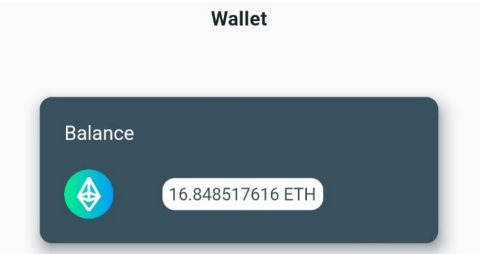

Figura 9.1: App wallet

Sicurezza In quanto vengono gestite criptovalute, l'utilizzo dell'app è securizzato tramite l'impronta digitale memorizzata nello smartphone. Questo al fine di evitare abusi e spese indesiderate.

Configurazione gateway Come descritto in precedenza, l'app consente di configurare i gateway attraverso comunicazioni bluetooth.

Gestione termostato Ovviamente l'app consente di gestire le funzionalità del termostato quali il cambio della temperatura di soglia e l'accensione/spegnimento. Consente anche di monitorare le temperature fornite dai sensori per ogni stanza e lo stato attuale dei sistemi di riscaldamento [Figura 9.2].

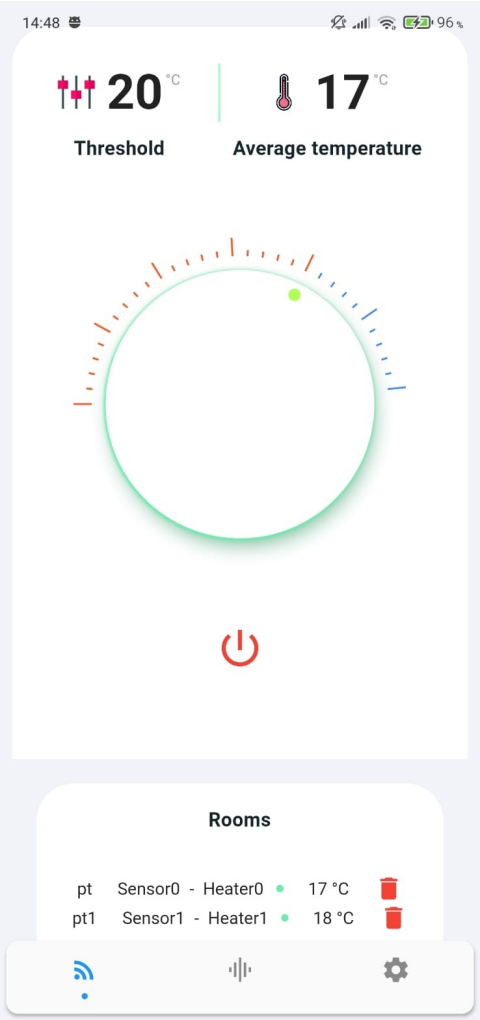

Figura 9.2: Gestione termostato app

9.2 Tecnologie

L'app è stata realizzata utilizzando le seguenti tecnologie:

Flutter Flutter è un framework per Dart creato recentemente da Google che consente la realizzazione di app mobile con UI nativa per Android e iOS. Recentemente gli sviluppatori di Flutter hanno introdotto la possibilità di realizzare anche siti web. Flutter, a differenza di molti framework, non utilizza webview con il vantaggio di ottenere prestazioni elevatissime. Inoltre, consente anche di eseguire codice nativo Java/Kotlin su Android e Swift su iOS.

Provider La gestione dello stato dell'applicazione è realizzata attraverso il pattern Provider. Inizialmente si è pensato di utilizzare Redux per la sue funzioni avanzate ma per la natura dell'app, questo era eccessivo e si è scelto di utilizzare Provider. Questo consente di creare uno stato globale e di rieseguire il build solamente dei widget dipendenti da tale stato quando vi è un aggiornamento di stato.

Flare Le animazioni sono state gestite grazie al framework Flare che consente di gestire animazioni 2D vettoriali. I file delle animazioni sono stati generati grazie al tool online Rive.

Web3Dart Come già citato in precedenza, per la comunicazione con la blockchain è stato utilizzato un porting della libreria web3 per il linguaggio Dart. Tuttavia questo porting è incompleto e per alcune cose è stato necessario creare delle transazioni manualmente, come ad esempio per il deploy dei contratti.

REST La comunicazione tra app mobile e server ausiliari avviene tramite il paradigma REST utilizzando massivamente JSON per lo scambio di entità.

9.3 Sequenza azioni

La completa funzionalità del sistema termostato si basa sull'internazione con l'app. La sequenza delle azioni per l'inizializzazione del tutto è in figura 9.3. Una volta inizializzato tutto, le interazioni saranno basate sui contratti di cui è stato effettuato il deploy in blockchain. Dunque, ogni interazione dell'utente con l'app produrrà una transazione che verrà letta tramite gli event handler dei gateway.

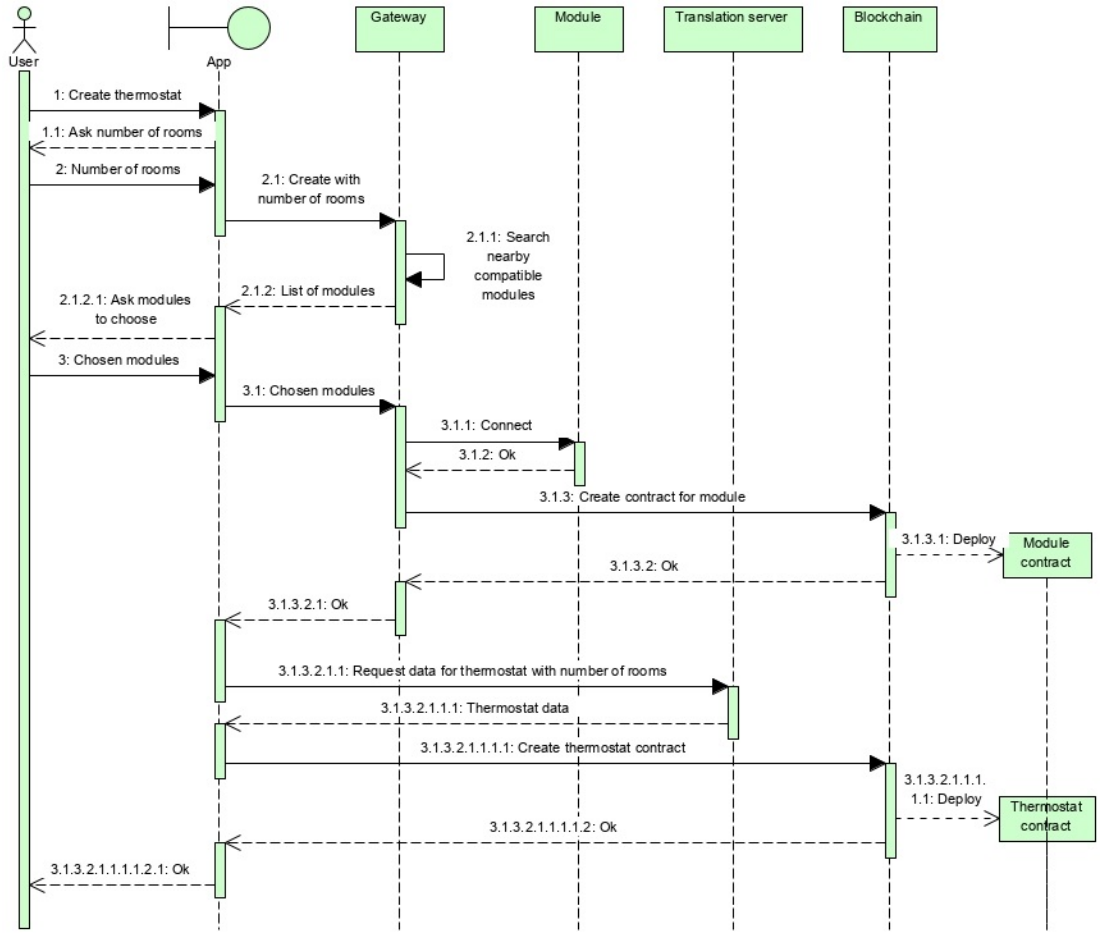

Figura 9.3: Sequence diagram informale di inizializzazione

10. Conclusioni e Sviluppi Futuri

Questo progetto è sicuramente migliorabile e contiene numerosi spunti per sviluppi futuri. La sua natura modulare ne semplifica il processo e fornisce componenti riutilizzabili per altri lavori.

10.1 Solcraft

La libreria Solcraft potrebbe essere espansa aggiungendo tutti i componenti del linguaggio Solidity in modo preciso. Inoltre, si possono introdurre numerosi design pattern come il Builder per semplificare ancor di più la creazione del codice e il Decorator.

10.2 Diagrammi

Il modello BPMN del termostato potrebbe essere certamente migliorato aggiungendo nuove funzionalità e migliorando quelle già presenti.

10.3 Traduttore

Il traduttore può essere migliorato aggiungendo anche un traduttore per diagrammi di collaborazione e migliorando la sintassi dei messaggi. Ad esempio, ci si è resi conto che il nome del messaggio è superfluo in quanto si ha già il nome del task. Si potrebbero supportare nella traduzione anche altre caratteristiche dei modelli BPMN. Si potrebbe anche rivedere l'algoritmo di traduzione migliorandone le prestazioni e la pulizia del codice.

10.4 Costi

Per il progetto è stata utilizzata la rete di test Rinkeby ma, senza dubbio, questo termostato è molto costoso in termini di gas e quindi anche in termini economici. Si evidenzia la necessità di un'analisi approfondita dei costi e di migliorare il traduttore al fine di ridurre il più possibile l'onere delle operazioni.

10.5 Modeler

Il naturale proseguimento del traduttore è quello di creare un modeler apposito simile a quello di ChorChain che supporti le sintassi descritte precedentemente.

Risorse

Repositories

sol-compiler [NodeJS - Express] Server per compilare codice Solidity
REPO: github.com/EmmanueleBollino/sol-compiler

bpmn2sol [Java] Traduttore da modello di coreografia a contratti Solidity
REPO: github.com/EmmanueleBollino/bpmn2sol

bpmn2sol-server [Java - Spring Boot - Spring Web] Server per bpmn2sol
REPO: github.com/EmmanueleBollino/bpmn2sol-server

solcraft [Java] Libreria per la generazione di codice Solidity
REPO: github.com/EmmanueleBollino/solcraft
MAVEN CENTRAL: search.maven.org/artifact/com.github.EmmanueleBollino/solcraft

ControlGateway [Python] Script per il software dei gateway
REPO: github.com/BackCamino/ControlGateway

ControlModules [Arduino] Codice per il funzionamento dei moduli sensore/heater
REPO: github.com/BackCamino/ControlModules

EthereumThermostatApp [Dart - Flutter] App per la gestione del termostato
REPO: github.com/diegodiome/EthereumThermostatApp

Multimedia

Presentazione [Powerpoint] Presentazione progetto
tinyurl.com/ethereumThermostatPresentation

Demo [Video] Demo del funzionamento del progetto
tinyurl.com/ethereumThermostatDemo

Relazione [PDF] Relazione del progetto
tinyurl.com/ethereumThermostatReport