Monitoring a CPS through a BigChainDB verified Finite State Machine

Enrico Catalfamo

November 25, 2023

Contents

Chapter 1

Introduction

La seguente relazione si pone l'obiettivo di illustrare e descrivere nel dettaglio il processo che ha portato alla creazione di un sistema tramite il quale è possibile monitorare il comportamento e l'andamento, nel tempo, di un dispositivo cyberfisico, tramite l'impiego di una macchina a stati finiti (che schematizza il device) e una blockchain-app basata su BigChainDB e Tendermint (che si occupa di verificare e validare gli stati attraversati dal dispositivo, nonché dalla FSM). Lo scopo, quindi, è quello di realizzare un digital twin (seppur grezzo) del dispositivo, che consenta, in futuro, di implementare funzionalità quali diagnostica, controllo e predizione del comportamento del dispositivo. Per semplicità, ma senza perdita di generalità, è stato preso, come device di riferimento, un termostato, la cui FSM consta di due stati e 4 possibili transizioni. La rete blockchain consta di 4 nodi: il nodo principale, realizzato tramite una macchina con Ubuntu 20.04, e 3 nodi secondari, realizzati tramite container docker. Al fine di inserire la funzione di validazione delle transizioni della FSM, è stato adottato un approccio differente per il nodo principale e i nodi secondari:

- Sul nodo principale è stata realizzata un'applicazione Flask, servita tramite uWSGI (dietro un server NGINX), che espone degli endpoint per:

  - Ottenere i dati relativi al dispositivo cyberfisico (dati che, in un contesto reale, verrebbero prelevati tramite sensori e costituirebbero gli input del dispositivo) e, quindi, calcolare la transizione di stato (se esiste) sulla FSM. Una volta calcolata e validata la transizione, questa viene inserita nella blockchain;
  - Restituire una pagina web tramite la quale è possibile visualizzare i dati relativi alla FSM (ingressi, stati, uscite);
  - Ottenere i dati relativi alla FSM in un dato intervallo di tempo (tramite i quali viene popolata la pagina web).

- Sui nodi secondari, invece, sono installati solo BigChainDB e Tendermint. Tuttavia, dal momento che BigChainDB, di default, non offre funzionalità di verifica delle transizioni per macchine a stati finiti, è stata realizzata una versione "adattata" del software per permettere anche ai nodi secondari di verificare le transizioni durante la fase di validazione delle transazioni.

Il codice e, in generale, tutto il materiale realizzato, verrà esplicato in seguito ed è reperibile nelle seguenti repository:

- `https://github.com/enrikata/BigChainDB-FSM---CPS-monitoring` (BDB application repository)
- `https://github.com/enrikata/bigchaindb_fsm` (FSM-adapted BigChainDB repository)

Chapter 2

Creating the FSM

Come abbiamo anticipato, il dispositivo di riferimento è un termostato. Dal momento che BigChainDB è scritto principalmente in Python, quest'ultimo è il nostro linguaggio di riferimento. Per tale motivo, al fine di realizzare la FSM associata al device, abbiamo utilizzato il modulo Python pytransitions. Tutta la documentazione relativa al modulo è reperibile al seguente link: `https://github.com/pytransitions/transitions`.
Pytransitions permette di definire un modello (il nostro device) come classe Python, e passarlo, successivamente, all'inizializzatore della classe Machine (che viene fornita dal modulo pytransitions e implementa le funzionalità delle macchine a stati) per "trasformarlo" nella corrispondente macchina a stati. Ne consegue che, nel codice, viene effettuato l'accesso a metodi e variabili che non sono effettivamente definiti, poiché verranno in seguito ereditati dalla classe Machine. Per capire meglio questo concetto, andiamo ad analizzare il codice del nostro modello:

```python
class FSM:

    def __init__(self):
        self.states = {'heating':{'heating_rate':1}, 'cooling':{'heating_rate':0}}
        self.initial_state = list(self.states.keys())[0]
        self.inputs = {'temperature':0}
        self.transitions = [
            {'trigger': 'heat', 'source': ['heating', 'cooling'], 'dest': 'heating', 'conditions':'check_low_temp'},
            {'trigger': 'cool', 'source': ['heating', 'cooling'], 'dest': 'cooling', 'conditions':'check_high_temp'}
        ]

    def check_low_temp(self):
        if self.inputs['temperature'] < 18:
            return True
        else:
            return False

    def check_high_temp(self):
        if self.inputs['temperature'] > 22:
            return True
        else:
            return False
```

Figure 2.1: FSM class definition

All'interno della funzione __init__, vengono definite le variabili d'istanza della classe. Possiamo notare che le definizioni delle variabili sono del tutto coerenti con i concetti relativi alle macchine a stati:

- Abbiamo un dizionario contenente gli stati attraversabili (e gli output del device nei relativi stati);
- Abbiamo uno stato iniziale (che, nel nostro caso, è lo stato "heating");
- Abbiamo gli input "attuali" del sistema (Nel caso del termostato, l'input è la temperatura, che triggera i cambiamenti di stato ed è inizialmente settata a 0);
- Abbiamo un dizionario che contiene le possibili transizioni, secondo lo schema implementato da pytransitions. Il valore associato alla chiave "trigger" definisce il nome della funzione che, una volta trasformato il modello in macchina a stati, avremo a disposizione per "triggerare", appunto, la transizione di stato; il valore associato alla chiave "source" definisce la lista di stati a partire dai quali è possibile effettuare la transizione; il valore associato alla chiave "dest" definisce lo

stato di arrivo della transizione; il valore associato alla chiave "conditions" definisce le condizioni da verificare affinché la transizione sia fattibile. Solitamente, il valore associato a conditions corrisponde al nome della funzione (definita nel modello) che si occupa di verificare le condizioni. Di fatto, è possibile notare, in figura, che le condizioni definite corrispondono ai due metodi implementati nella classe, che fanno un check sulla temperatura rilevata dal termostato.

Questi elementi costituiscono, in sostanza, lo scheletro della nostra FSM. Tuttavia, abbiamo bisogno di altri metodi che implementino funzionalità ulteriori. Prima di proseguire con la descrizione del codice, è opportuno definire alcuni concetti chiave per la compresione di quanto è stato fatto.

Chapter 3

How does BigChainDB store data?

BigChainDB si appoggia a MongoDB per effettuare lo storage vero e proprio dei dati. Tendermint, invece, realizza l'algoritmo di consenso, che è fondamentale in una blockchain, costituita da più nodi che devono validare i dati (e quindi concordare su essi). Per strutturare i dati, BigChainDB utilizza gli asset, che possono teoricamente rappresentare qualunque cosa (un oggetto fisico, un oggetto digitale, ecc.). Tali asset possono essere creati e/o trasferiti: perciò, l'asset implica indirettamente anche il concetto di possesso. L'appartenenza dell'asset è generalmente attribuita al possessore della chiave privata con cui la transazione è stata firmata prima essere validata dai nodi e registrata sul database. A più basso livello, i dati vengono strutturati come JSON. Una transazione, dunque, è costituita da un JSON contenente diversi campi, a cui sono associati diversi valori. Nel nostro caso, dunque, dobbiamo decidere come rappresentare le informazioni relative alle transizioni della macchina a stati. Il modello che è stato impiegato è il seguente:

```
    'data':{
        'entity': 'fsm',
        'from':initial_state,
        'to': final_state,
        'input' : self.inputs,
        'output': self.get_output()
    }
}
```

Figure 3.1: FSM class definition

Di seguito, una descrizione del contenuto del modello:

- la chiave "entity" è stata impiegata solo e unicamente per facilitare la ricerca delle transazioni su BDB tramite il corrispettivo driver Python (che permette, come qualunque altro driver, di connettersi al database ed effettuare query) ed il suo valore, per tale motivo, è sempre "fsm" per tutte le transazioni;
- la chiave "from" contiene lo stato di partenza, a partire dal quale è stata effettuata la transizione;
- la chiave "to" contiene lo stato di arrivo, a cui si è giunti per mezzo della transizione;
- il valore associato alla chiave "input" è un dizionario contenente, come chiavi, i "nomi" degli ingressi al device, e, come valori, i valori di tali ingressi che hanno generato la transizione di stato;
- il valore associato alla chiave "output", esattamente come per la chiave "input", è un dizionario che contiene gli output del device legati al nuovo stato.

In più, anche se non è mostrato in figura, il modello contiene una chiave "timestamp" in cui sono inserite data e ora della creazione della transazione. Per maggiori insights su BigChainDB, assets e transazioni, può essere utile

consultare i seguenti link: https://www.bigchaindb.com/developers/guide/key-concepts-of-bigchaindb/, https://docs.bigchaindb.com/en/latest/about-bigchaindb.html .

Chapter 4

Extending the FSM

Adesso che abbiamo qualche nozione circa il modo in cui BigChainDB conserva i dati e il modello di transazione che intendiamo utilizzare, possiamo proseguire con la descrizione della creazione della nostra macchina a stati. Chiaramente, Vogliamo che la nostra macchina a stati possa:

- essere inizializzata a partire dai dati contenuti su BigChainDB;
- restituire un dizionario contenente l'output nello stato corrente;
- verificare l'esistenza di una transizione dallo stato attuale a un altro a partire dagli input;
- aggiornare gli input a partire da un dict contenente i nuovi valori.

Tutte queste funzioni sono state implementate e sono visualizzabili nell'immagine seguente.

```python
def get_output(self, in_state = None):
    if in_state == None:
        in_state = self.state
    return self.states[in_state]

def get_transition(self):
    for transition in self.transitions:
        func = getattr(self, 'may_' + transition['trigger'])
        if func():
            return getattr(self, transition['trigger'])
    return None

def get_input_keys(self):
    return list(self.inputs.keys)

def update_inputs(self, dict):
    if self.inputs.keys() == dict.keys():
        flag = 0
        for key, value in dict.items():
            if not isinstance(value, type(self.inputs[key])):
                flag = 1
        if flag == 0:
            self.inputs = {k: dict[k] for k in self.inputs}
            return True
    return False

def initialize_fsm(self, init_dict):
    self.update_inputs(init_dict['data']['input'])
    init_state = init_dict['data']['to']
    transition_func = getattr(self, 'to_' + init_state)
    transition_func()
```

Figure 4.1: The other functions defined in the FSM class

Il loro funzionamento è abbastanza semplice:

- La funzione get_output() restituisce l'output relativo allo stato inserito in ingresso come parametro. Se non viene inserito, restituisce l'output nello stato corrente della FSM;

- La funzione get_transition() preleva i trigger dal dict delle transizioni e verifica l'esistenza della funzione "may_trigger". Tale funzione verrà "implementata automaticamente" una volta che avremo trasformato il nostro modello in FSM tramite pytransitions, e restituisce True o

False a seconda del fatto che la transizione relativa a quel trigger sia fattibile o meno. Quindi, se è fattibile, viene restituita al chiamante direttamente la funzione trigger, per effettuare la transizione;

- La funzione update_inputs() preleva il dizionario dai parametri di ingresso, verifica che le chiavi e il tipo di dato dei valori combacino con quelle del modello e, in caso affermativo, aggiorna la variabile di istanza degli input e restituisce True;

- La funzione initialize_fsm(), a partire da un dizionario come quello in figura 3.1, inizializza gli input e lo stato attuale coi valori in esso contenuti.

Infine, ci servono due funzioni: una che effettui il "running" vero e proprio della macchina a stati e una che verifichi semplicemente la validità e fattibilità di una transizione, senza, di fatto, effettuarla. I due metodi sono visualizzabili nell'immagine seguente.

```
def run_fsm(self, input_dict):
    initial_state=self.state
    final_state=self.state
    self.update_inputs(input_dict)
    transition = self.get_transition()
    if transition is not None:
        transition()
        final_state=self.state
    output = self.get_output()

    fsm_info = {
        'data':{
            'entity': 'fsm',
            'from':initial_state,
            'to': final_state,
            'input' : self.inputs,
            'output': self.get_output()
        }
    }

    return fsm_info


def verify_transition(self, info_dict):
    init_state = info_dict["data"]["from"]
    final_state = info_dict["data"]["to"]
    inputs = info_dict["data"]["input"]
    to_init_state = getattr(self, 'to_' + init_state)
    to_init_state()
    self.update_inputs(inputs)
    to_final_state = getattr(self, 'may_' + final_state)
    if to_final_state():
        return True
    else:
        return False
```

Figure 4.2: The two methods to run and verify a transition

La funzione run_fsm() è il cuore pulsante della classe che abbiamo definito. Di fatto, è l'unico metodo che viene richiamato all'interno della nostra applicazione BDB, in quanto effettua tutte le operazioni necessarie ai nostri scopi. Di fatto:

- crea due variabili, initial_state e final_state, inizialmente settate con lo stesso valore (lo stato corrente);

- utilizza il metodo update_inputs(), definito precedentemente, per aggiornare i valori degli ingressi con quelli contenuti nel dizionario input_dict, parametro di input della funzione;
- utilizza il metodo get_transition() per ottenere, se possibile, il trigger (metodo) necessario per effettuare la transizione;
- richiama, se esiste, il trigger, effettuando la transizione di stato e settando la variabile final_state con lo stato raggiunto;
- utilizza il metodo get_output() per calcolare l'output del modello nel nuovo stato;
- Restituisce al chiamante un dict con tutte le informazioni relative alla transizione. È importante notare che, se nessuna transizione è realizzabile, verrà restituito un dizionario con stato di partenza e di arrivo coincidenti. Ciò ha senso, poiché potrebbe essere di nostro interesse monitorare l'evoluzione degli ingressi quand'anche non vi sia una transizione di stato.

Il nostro modello è finalmente completo. Se volessimo ricavarne una macchina a stati, dovremmo semplicemente importare la classe FSM, la classe Machine da Pytransitions e scrivere le seguenti righe di codice:

```
fsm = FSM()
thermostat = Machine(model=fsm,
↪  states=list(fsm.states.keys()), transitions =
↪  fsm.transitions, initial=fsm.initial_state))
```

Successivamente, sarà possibile "controllare" la nostra FSM direttamente tramite l'oggetto fsm, che acquisirà i metodi della classe Machine.

Chapter 5

Creating the BigChainDB app

Per creare l'applicazione BigChainDB, come anticipato, è stato utilizzato Flask.

Flask è un microframework web per la creazione di webapp in Python, ed è progettato per essere semplice, leggero e facile da estendere. Permette di creare un'applicazione in pochi passaggi, fornendo dei metodi di interfaccia per l'handling delle richieste e delle risposte HTTP e sfruttando il concetto di routing per associare le richieste HTTP a funzioni specifiche. La struttura, a livello di files e directories, della nostra applicazione è la seguente:

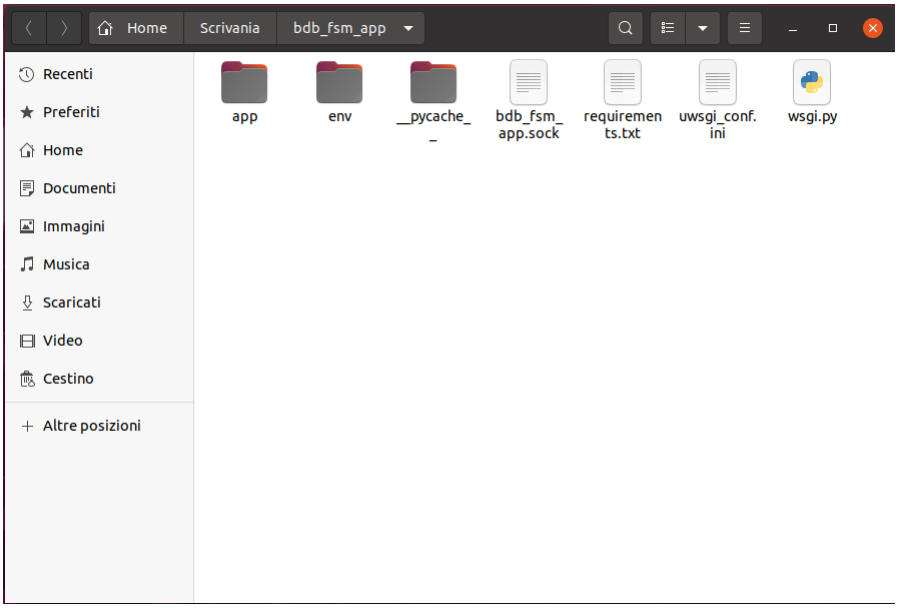

Figure 5.1: The application main directory

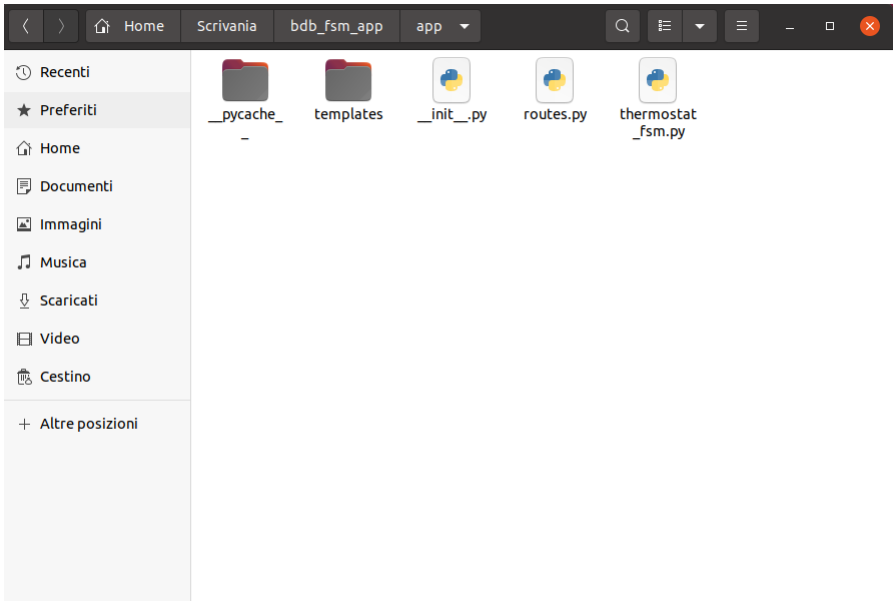

Figure 5.2: The app subdirectory

Cominciamo dalla main directory:

- il file wsgi.py importa il modulo definito nella directory app. L'import,

in automatico, esegue il codice contenuto nel file __init__ del modulo;

- il file uwsgi_conf.ini contiene i parametri di configurazione di uwsgi per l'applicazione. Per maggiori insights, è possibile consultare il link `https://www.bloomberg.com/company/stories/configuring-uwsgi-production-deploymen`

- il file requirements.txt contiene tutti l'elenco di tutti i moduli Python di cui l'applicazione necessita. Tali moduli vanno installati all'interno di un Python virtual environment. Per maggiori insights circa i virtualenv, è possibile consultare il seguente link: `https://docs.python.org/3/tutorial/venv.html`;

- il file bdb_fsm_app.sock è una socket UNIX, creata in automatico da uWSGI nel momento in cui viene startato. Attraverso questa socket, uWSGI riceve i dati da NGINX e li inoltra all'applicazione, e viceversa.

Per quanto riguarda la subdirectory app, invece:

- Il file thermostat.py contiene la definizione della classe FSM, di cui abbiamo parlato nei capitoli precedenti;

- Il file routes.py contiene la logica Flask, nonché la definizione degli endpoint e delle funzioni da eseguire quando vengono ricevute richieste su questi ultimi, e importa la classe FSM per creare una macchina a stati finiti (che evolve durante l'esecuzione dell'applicazione);

- Il file __init__.py contiene la logica da eseguire nel momento in cui il modulo app viene importato, nonché la creazione di un Flask application object e l'import del file routes.py.

Tutto ciò che è di nostro interesse è definito nel file routes.

```
bdb_root_url = 'http://localhost:9984'
bdb = BigchainDB(bdb_root_url)
keys = generate_keypair()

collection = bdb.assets
transactions = bdb.transactions

fsm = FSM()

thermostat = GraphMachine(model=fsm, states=list(fsm.states.keys()), initial=fsm.initial_state, transitions=fsm.transitions)

#### FSM initialization ####
last_state = collection.get(search="fsm")

if len(last_state) > 0:
    last_state = last_state[-1]
    fsm.initialize_fsm(last_state)
#### FSM initialization ####
```

Figure 5.3: First lines of code in routes.py

In figura, sono presenti le prime righe di codice del file (import esclusi), in cui:

- viene creata un'istanza di connessione a BigChainDB;
- viene generata una coppia di chiavi pubblica e privata (che l'applicazione utilizzerà per firmare le transazioni);
- vengono creati due oggetti di interfaccia agli assets e alle transazioni definite su BigChainDB;
- viene creata un'istanza della classe FSM e associata a una GraphMachine (identica a una Machine semplice, ma con funzionalità che permettono di ottenere un'immagine del grafo della macchina a stati);
- Vengono prelevati tutti gli asset, ovvero tutte le transizioni della FSM presenti su BDB;
- Viene inizializzata la macchina a stati con i dati contenuti nell'ultimo asset creato (che, ovviamente, contiene i dati più recenti).

Successivamente, l'applicazione definisce gli endpoint e la logica sottostante. L'endpoint che più ci interessa è /data, ovvero quello tramite

cui il nodo riceve i dati dai sensori del dispositivo fisico e aggiorna lo stato della FSM in maniera concorde.

```
@app.route("/data", methods=['POST'])
def create_transaction():
    data = json.loads(request.data)
    transaction_data = fsm.run_fsm(data)
    transaction_data["data"]["timestamp"] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    prepared_creation_tx = bdb.transactions.prepare(operation='CREATE', signers=keys.public_key, asset=transaction_data)
    fulfilled_creation_tx = bdb.transactions.fulfill(prepared_creation_tx, private_keys=keys.private_key)
    sent_creation_tx = bdb.transactions.send_commit(fulfilled_creation_tx)
    return '{}'.format(json.dumps(sent_creation_tx))
```

Figure 5.4: /data endpoint definition

Il codice non è complesso, in quanto la maggior parte del lavoro viene svolto dalla classe FSM. La funzione create_transaction si limita a:

- prelevare i dati (gli input della fsm, in un JSON) dalla richiesta ricevuta e caricarli in un dict;
- richiamare la funzione run_fsm() per runnare la macchina a stati;
- inserire, nel dizionario di output della suddetta funzione, un campo "timestamp" con data e ora attuali;
- creare una transazione BDB;
- firmare la transazione con la chiave privata precedentemente creata;
- inviare la transazione agli altri nodi BDB e attendere che il nuovo stato venga committato da tutti;
- restituire al chiamante il JSON relativo alla transazione effettuata.

Per quanto riguarda gli altri due endpoint, invece, /graph restituisce una pagina html su cui è possibile visualizzare il grafo della FSM e i dati relativi a input, output e stati attraversati; /graphimages, invece, restituisce quest'ultimi grafici sotto forma di immagini png codificate in base64. I dati rappresentati nelle immagini vengono prelevati dall'applicazione facendo una query su BigChainDB e filtrando gli asset in base al timestamp.

```python
@app.route("/graphimages", methods=["GET"])
def get_graph_images():

    start_date = request.args.get('startDate')
    end_date = request.args.get('endDate')

    if start_date != None and end_date != None and start_date <= end_date:

        json_response = {}

        input_labels = []
        output_labels = []

        ### FSM image ###
        b = io.BytesIO()
        fsm.get_graph().draw(b, format="png", prog='dot')
        b.seek(0)
        fsm_img_str = base64.b64encode(b.read()).decode('utf-8')
        json_response["fsm_img_str"] = fsm_img_str
        ### FSM image ###

        transitions = collection.get(search="fsm")

        transitions = [transition for transition in transitions if transition['data']['timestamp'] >= start_date and transiti

        ### Get labels for legends in graphs ###
        if len(transitions)>0:
            data_model = transitions[-1]
            input_labels = data_model['data']['input'].keys()
            output_labels = data_model['data']['output'].keys()

        ### Get labels for legends in graphs ###

        inputs = [list(entry['data']['input'].values()) for entry in transitions]
        outputs = [list(entry['data']['output'].values()) for entry in transitions]
        state_trans = [entry['data']['to'] for entry in transitions]
        timestamps = [entry['data']['timestamp'] for entry in transitions]

        timestamps = [datetime.strptime(timestamp, '%Y-%m-%d %H:%M:%S') for timestamp in timestamps]
```

Figure 5.5: /graphimages definition

Più nello specifico, quando il client effettua una GET sull'endpoint /graphimages, ciò che accade lato applicazione è quanto segue:

- Vengono prelevati i parametri startDate ed endDate della GET (corrispondenti con gli estremi temporali dei dati da prelevare);
- Si verificano alcune condizioni su questi parametri (innanzitutto se esistono, dopodichè se la data di inizio è minore o uguale alla data di fine);
- Si crea un dict vuoto, che conterrà il payload della risposta dell'applicazione (le immagini su cui sono graficati i dati);

- Si creano due liste vuote, che conterranno le labels (i nomi, per intenderci) dei parametri di input e output della FSM (saranno utili alla creazione delle legende all'interno dei grafici);
- Si crea un buffer di bytes in memoria
- si ricava l'immagine della FSM tramite i metodi di interfaccia della classe GraphMachine e la si inserisce nel buffer;
- Si ottiene una rappresentazione in base64 dei dati;
- Si inserisce tale rappresentazione nel dizionario da restituire;
- Si prelevano tutte le transazioni sul database;
- Si filtrano in base al timestamp, per ottenere le transizioni di stato nell'intervallo temporale di interesse;
- Si prelevano le labels dal data model, prendendo, come riferimento, l'ultima transazione restituita da BigChainDB;
- Si separano le informazioni contenute nelle transazioni in liste differenti (una per gli input, una per gli output, una per le transizioni di stato), in maniera tale da poter generare grafici agevolmente
- Si convertono le stringhe timestamp in oggetti di tipo datetime (in maniera tale da poterle utilizzare per generare grafici con MatPlotLib);
- Si creano i grafici relativi a input, output e transizioni di stato con le stesse modalità viste per l'immagine della FSM (l'unica differenza è, appunto, l'utilizzo di MatPlotLib per la generazione);
- Si restituisce il dizionario (come JSON) con tutti i dati.

Abbiamo descritto le dinamiche e le funzionalità inerenti all'applicazione. Nei capitoli successivi vedremo come installare le componenti essenziali del sistema e come creare i nodi secondari della blockchain.

Chapter 6

Installing and setting up uWSGI

uWSGI è una implementazione specifica di un server WSGI (Web Server Gateway Interface) tipicamente utilizzata per runnare applicazioni web Python. Opera come un ponte tra un server Web (come Nginx o Apache) e un'applicazione Web scritta in un linguaggio di scripting supportato.
uWSGI va installato tramite pip (`pip install uwsgi`), all'interno del virtual environment Python creato per l'applicazione. Affinché possa essere gestito da systemd (e quindi controllato tramite `systemctl`), è necessario creare un service file associato. A tal fine, navighiamo nella folder /etc/systemd/system e creiamo un file col nome `uwsgi.service`. Il suo contenuto sarà il seguente:

```
[Unit]
Description=BDB FSM monitoring app uWSGI service
[Service]
User=enrikata
Group=www-data
Environment=PATH=/home/enrikata/Scrivania/bdb_fsm_app/env/bin:/usr/bin
WorkingDirectory=/home/enrikata/Scrivania/bdb_fsm_app
ExecStart=/home/enrikata/Scrivania/bdb_fsm_app/env/bin/uwsgi --ini uwsgi_conf.ini
Restart=always
[Install]
WantedBy=multi-user.target
```

Figure 6.1: uwsgi.service file content

Di seguito, una descrizione del contenuto:

- `Description` contiene una descrizione del servizio;
- `User` e `Group` definiscono sotto quale utente e gruppo il servizio verrà eseguito;
- `Environment` definisce le variabili d'ambiente necessarie per il corretto funzionamento del servizio (nel nostro caso, viene settata la variabile d'ambiente PATH con la cartella /bin del virtualenv relativo all'applicazione);
- `WorkingDirectory` Definisce in quale directory il servizio sarà avviato;
- `ExecStart` Definisce quali comandi eseguire quando il servizio viene lanciato. In questo caso, viene richiamato uwsgi dal virtual environment, passandogli il file di configurazione uwsgi_conf.ini;
- `Restart` Definisce la policy di restart del servizio. In questo caso, verrà tentato il restart del servizio ogni qualvolta dovesse andare down;
- `WantedBy` Definisce le modalità con cui il servizio deve essere avviato automaticamente.

Chapter 7

Installing and setting up NGINX

Nginx è un software open source principalmente utilizzato per il web serving, reverse proxing e load balancing. È possibile installarlo mediante il package manager apt (`sudo apt install nginx`). Una volta installato, sarà avviato in automatico. Per modificare il file di configurazione, è necessario spostarsi nella directory `/etc/nginx`, al cui interno è presente il file `nginx.conf`. Al suo interno, sopra la direttiva `include /etc/nginx/conf.d/*.conf;`, sarà necessario inserire quanto segue:

```
server {
        listen 80;
        server_name 192.168.1.9;
        root /home/enrikata/Scrivania/bdb_fsm_app;
        location / {
                include uwsgi_params;
                uwsgi_pass unix:/home/enrikata/Scrivania/bdb_fsm_app/bdb_fsm_app.sock;
                }
}

include /etc/nginx/conf.d/*.conf;
include /etc/nginx/sites-enabled/*;
```

Figure 7.1: NGINX configuration file content

In questo modo, istanzieremo un server HTTP sulla porta 80 e sull'IP

definito da `server_name` (in questo caso utilizziamo un IP privato perché la scheda di rete della macchina virtuale è settata in bridge mode. In questo modo, potremo accedere all'applicazione da qualunque dispositivo sulla stessa rete locale). La direttiva `root` definisce la directory di root che verrà utilizzata per cercare e restituire un file, qualora venisse richiesto (in realtà non serve a nulla per i nostri scopi). la direttiva `location /` definisce il comportamento che il server deve assumere quando l'URI della richiesta inizia per "/" (dunque, per qualunque richiesta). Poiché vogliamo che tutte le richieste vengano passate a uWSGI, inseriamo la direttiva `uwsgi_pass` seguita dal path della sua socket UNIX. In questo modo, NGINX farà da reverse proxy, reindirizzando tutte le richieste a uWSGI.

Chapter 8

Modifying BigChainDB source code

Come abbiamo anticipato, la logica di verifica delle transizioni sulla FSM è inserita, sul nodo coordinatore, nell'applicazione Flask. Si potrebbe pensare che una tale soluzione sia applicabile anche sui nodi secondari della blockchain; tuttavia, non è così. Di fatto, lo scambio e la validazione delle transazioni tra i nodi della blockchain sono mediati da Tendermint, che dialoga con BigChainDB mediante l'interfaccia ABCI (Application BlockChain Interface). La conseguenza di ciò è che, quand'anche l'applicazione dovesse essere installata sui nodi secondari, le transazioni scambiate non "salirebbero" fino a raggiungere l'applicazione, per cui la validazione consterebbe solo nel check della correttezza del formato JSON e degli elementi crittografici della transazione. Per tale motivo, sui nodi secondari è necessario installare una versione di BigChainDB che si presti alla validazione del contenuto della transazione, che, nel nostro caso, è una transizione di stato sulla FSM. Fortunatamente, abbiamo precedentemente predisposto tutti gli elementi necessari per la verifica delle transizioni della FSM.

Andiamo ad analizzare la struttura del codice sorgente di BigChainDB. Per prima cosa, scarichiamo la repository github, reperibile al seguente link:

`https://github.com/bigchaindb/bigchaindb`.

nel file `bigchaindb/models.py` è definita la classe `Transaction`, contenente il metodo `validate`.

```
class Transaction(Transaction):
    ASSET = 'asset'
    METADATA = 'metadata'
    DATA = 'data'

    def validate(self, bigchain, current_transactions=[]):
        """Validate transaction spend
        Args:
            bigchain (BigchainDB): an instantiated bigchaindb.BigchainDB object.
        Returns:
            The transaction (Transaction) if the transaction is valid else it
            raises an exception describing the reason why the transaction is
            invalid.
        Raises:
            ValidationError: If the transaction is invalid
        """
        input_conditions = []

        if self.operation == Transaction.CREATE:
            duplicates = any(txn for txn in current_transactions if txn.id == self.id)
            if bigchain.is_committed(self.id) or duplicates:
                raise DuplicateTransaction('transaction `{}` already exists'
                                           .format(self.id))

            if not self.inputs_valid(input_conditions):
                raise InvalidSignature('Transaction signature is invalid.')

        elif self.operation == Transaction.TRANSFER:
            self.validate_transfer_inputs(bigchain, current_transactions)

        return self
```

Figure 8.1: The Transaction class defined in bigchaindb/models.py

Qui, inseriremo la logica di verifica della transizione. Per far ciò, inseriamo un modulo apposito. Creiamo la folder `bigchaindb/fsm_validation` e, al suo interno, inseriamo il file `thermostat_fsm.py` creato in precedenza. In più, inseriamo un file `__init__.py`, col seguente contenuto:

```
1 from transitions import Machine
2 from bigchaindb.fsm_validation.thermostat_fsm import FSM
3
4 fsm = FSM()
5
6 thermostat = Machine(model=fsm,
7                       states=list(fsm.states.keys()),
8                       initial=fsm.initial_state,
9                       transitions=fsm.transitions)
```

Figure 8.2: fsm_validation module __init__.py

Quindi, nella sezione degli import di `bigchaindb/models.py`, inseriamo la riga `from bigchaindb.fsm_validation import fsm`. Inoltre, modifichiamo la riga di import `from bigchaindb.common.exceptions import (InvalidSignature, DuplicateTransaction)` in `from bigchaindb.common.exceptions import (InvalidSignature, DuplicateTransaction, ValidationError)`. Finalmente, possiamo modificare la funzione `Transaction.validate()` come segue:

```
def validate(self, bigchain, current_transactions=[]):
    """Validate transaction spend
    Args:
        bigchain (BigchainDB): an instantiated bigchaindb.BigchainDB object.
    Returns:
        The transaction (Transaction) if the transaction is valid else it
        raises an exception describing the reason why the transaction is
        invalid.
    Raises:
        ValidationError: If the transaction is invalid
    """
    input_conditions = []

    if self.operation == Transaction.CREATE:
        duplicates = any(txn for txn in current_transactions if txn.id == self.id)
        if bigchain.is_committed(self.id) or duplicates:
            raise DuplicateTransaction('transaction `{}` already exists'
                                       .format(self.id))

        if not self.inputs_valid(input_conditions):
            raise InvalidSignature('Transaction signature is invalid.')

        if not fsm.verify_transition(self.asset):
            raise ValidationError('Transition from state `{}` to state `{}` with the

    elif self.operation == Transaction.TRANSFER:
        self.validate_transfer_inputs(bigchain, current_transactions)

    return self
```

Figure 8.3: Modified validate() function

La nuova funzione, assodato che la transazione sia di tipo `create` (non usiamo transazioni di tipo `transfer` nella nostra applicazione), verifica:

- Che la transazione non sia già stata registrata sulla blockchain (`bigchain.is_committed()`) né già presente nel mempool;
- Che gli elementi crittografici (hash e firma) siano validi;
- Che il contenuto della transazione, nonché la transizione di stato della FSM, sia valido.

Per concludere, modifichiamo il file `setup.py`, inserendo `pytransitions==0.9.0` nella lista `install_requires`.

Adesso, abbiamo tutto ciò che ci serve per creare l'immagine docker dei container che realizzeranno i nodi secondari. Questa versione modificata di

BigChainDB può essere installata col comando `python3 setup.py install` dalla main directory della repository.

Chapter 9

Setting up side nodes

Per settare i nodi secondari, dobbiamo:

- Realizzare un immagine docker con tutte le componenti necessarie;
- Configurare la rete BigChainDB/Tendermint per far si che tutti i nodi siano reciprocamente consapevoli della propria esistenza e possano connettersi per attuare i meccanismi di consenso;
- Realizzare un file docker compose per lanciare i nodi in contemporanea e avviare, sugli stessi, i servizi necessari.

Cominciamo dal primo step. Partendo da una container image di Ubuntu 20.04, vogliamo inserire MongoDB, BigChainDB e Tendermint. Il Dockerfile, dunque, dovrà essere il seguente:

```
FROM ubuntu:20.04
USER root
RUN apt-get update && apt-get install -y python3-pip libssl-dev && apt-get install -y mongodb && apt-get install -y git
RUN apt-get install -y unzip
RUN apt-get install -y wget
RUN wget https://github.com/tendermint/tendermint/releases/download/v0.31.5/tendermint_v0.31.5_linux_amd64.zip
RUN unzip tendermint_v0.31.5_linux_amd64.zip && rm tendermint_v0.31.5_linux_amd64.zip && mv tendermint /usr/local/bin
RUN git clone https://github.com/enrikata/bigchaindb_fsm
RUN cd bigchaindb_fsm && python3 setup.py install
RUN apt-get install -y ufw
RUN apt-get install -y sudo
RUN apt-get install nano
RUN useradd -m docker
RUN echo "docker:docker" | chpasswd && adduser docker sudo
RUN echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
RUN sudo ufw allow 26656
RUN bigchaindb -y configure
```

Figure 9.1: Dockerfile content

Come è possibile vedere in figura, il Dockerfile esegue tutti i comandi necessari all'installazione delle componenti del sistema, tra cui la nostra versione adattata di BigChainDB. Viene, inoltre, consentito il traffico sulla porta 26656, che Tendermint impiega per realizzare il p2p con gli altri nodi della rete. Tramite il comando `bigchaindb -y configure`, viene generato il file di configurazione di BigChainDB con i valori di default, che, per i nostri scopi, vanno più che bene.

Per quanto riguarda il secondo step, la directory `Docker` della repo `BigChainDB-FSM---CPS-monitoring` contiene tutti i file di configurazione di Tendermint necessari per connettere i nodi tra loro. Per maggiori informazioni circa la creazione di una rete BigChainDB, è possibile consultare il seguente link: `https://docs.bigchaindb.com/projects/server/en/latest/simple-deployment-template/network-setup.html`. Prima di realizzare il file docker compose, occorre creare una rete docker per far sì che i container e l'host possano raggiungersi a vicenda e si trovino sulla stessa rete (virtuale). Partendo dal presupposto che Docker sia installato (`https://docs.docker.com/engine/install/ubuntu/`), possiamo creare una rete bridge col seguente comando:

```
docker network create --driver=bridge --subnet=172.18.0.0/24 bdbnetwork
```

La rete ottenuta è taggata col nome `bdbnetwork`. Nel nostro file docker compose, quindi, definiremo 3 servizi (ogni servizio è un container) e gli assegneremo un IP nel range consentito da `bdbnetwork`. I servizi saranno tra loro identici, ovviamente, giacché devono effettuare le stesse operazioni. Di seguito, la definizione di uno dei 3 container (l'ultimo in ordine nel file docker-compose):

```
  Member3:
    image: bdb
    volumes:
      - ./member3/tendermint:/root/ftp
    command: sh -c "if [ ! -d /root/.tendermint ]; then mkdir /root/.tendermint &&
    cp -r /root/ftp/* /root/.tendermint; fi && service mongodb start &&
    nohup bigchaindb start & nohup tendermint node"
    networks:
      bdbnetwork:
        ipv4_address: 172.18.0.5


networks:
  bdbnetwork:
    external: true
```

Figure 9.2: Docker compose Member3 definition

Gli IP degli altri due container sono, rispettivamente, `172.18.0.3` e `172.18.0.4`. Ogni container usa l'immagine `bdb` (relativa al Dockerfile che abbiamo definito precedentemente) ed ha un volume associato (una directory del sistema host che viene montata sul container per condividerne il contenuto). Ogni container ha un volume a sé, dal momento che quest'ultimo contiene i file di configurazione di Tendermint che identificano il nodo. Nella direttiva `command`, vengono inseriti i comandi da eseguire all'avvio del container. Questi comandi sono gli stessi su tutti e 3 i nodi, ed effettuano le seguenti operazioni:

- Se non esiste la directory `.tendermint`, che contiene i file di configurazione e di stato di Tendermint, viene creata e, al suo interno, vengono

copiati i file presenti nel volume condiviso col sistema host (questo permette di configurare correttamente l'applicativo al primo avvio del container e garantire la coerenza tra lo stato di Tendermint e BigChainDB, nonché la persistenza dei dati, agli avvii successivi);

- Viene startato MongoDB;
- Viene startato BigChainDB;
- Viene startato Tendermint.

Chapter 10

Running the system

Per runnare il sistema, occorre:

- installare BigChainDB 2.2.2 e Tendermint sulla macchina host (`http://docs.bigchaindb.com/projects/server/en/latest/simple-deployment-template/set-up-node-software.html`);
- scaricare la repository `https://github.com/enrikata/BigChainDB-FSM---CPS-monitoring`;
- creare un virtualenv python all'interno della directory `bsb_fsm_app`;
- attivare il nuovo ambiente virtuale (`source env/bin/activate`);
- installare tutti i moduli elencati nel file `requirements.py` (`pip install -r requirements.py`);
- configurare uWSGI, come spiegato nei capitoli precedenti;
- installare e configurare NGINX, come spiegato nei capitoli precedenti;
- installare e configurare docker, come spiegato nei capitoli precedenti;
- passare nella directory `Docker` della repository;
- buildare l'immagine definita nel Dockerfile (`sudo docker build -f Dockerfile --tag bdb .`);

- tirare su i nodi tramite docker compose (`sudo docker compose up`);
- attendere che i nodi si sincronizzino, creando il blocco di start della blockchain (la creazione viene visualizzata sul terminale come log di Tendermint);

Una volta che il sistema è up, è possibile creare uno script per inviare richieste all'applicazione. Un esempio di script in javascript è il seguente:

```
const url = 'http://192.168.1.9/data';

const data = {
    temperature: 15
};

const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
};


fetch(url, options)
  .then(response => {
    if (response.ok) {
      return response.json();
    } else {
      console.log(response)
      throw new Error('Errore nella richiesta POST');
    }
  })
  .then(data => {
    console.log('Risposta del server:', data);
  })
  .catch(error => {
    console.error('Errore:', error);
  });
```

Figure 10.1: JavaScript code to perform request on the app /data endpoint

Chiaramente, l'IP deve essere modificato con quello relativo alla macchina

su cui gira l'applicazione. Dopo aver popolato BDB effettuando qualche richiesta, è possibile visualizzare i dati accedendo, tramite browser, all'endpoint /graph dell'app.

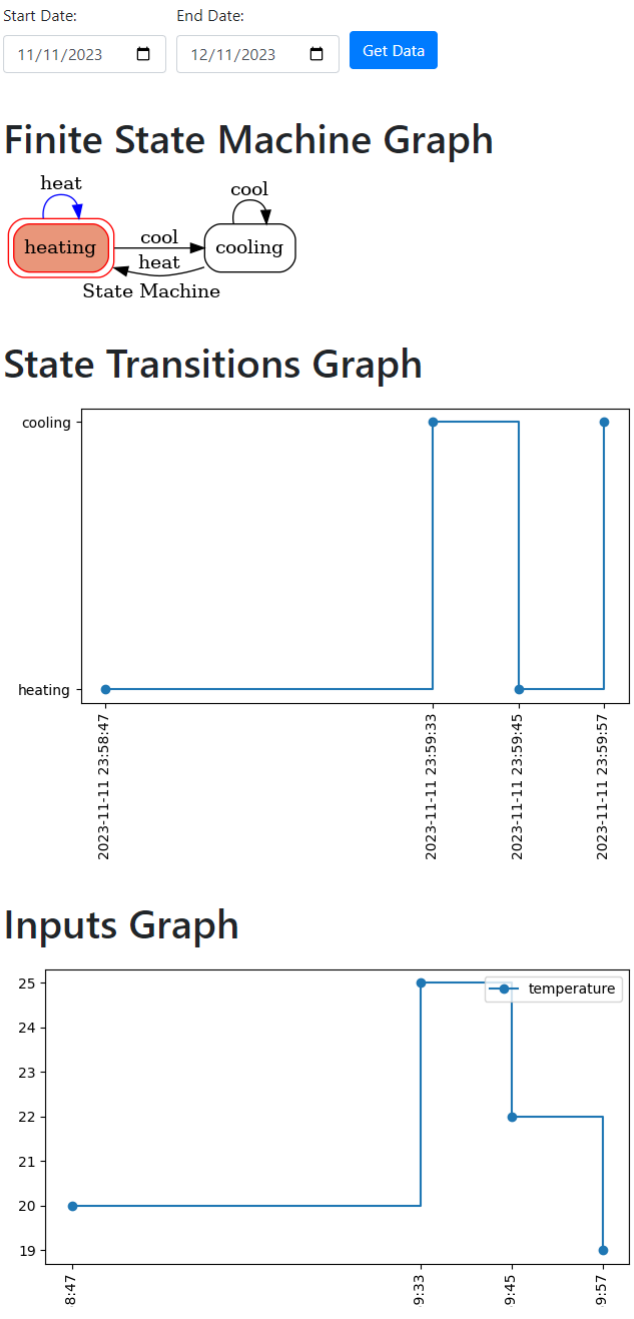

Figure 10.2: /graph endpoint