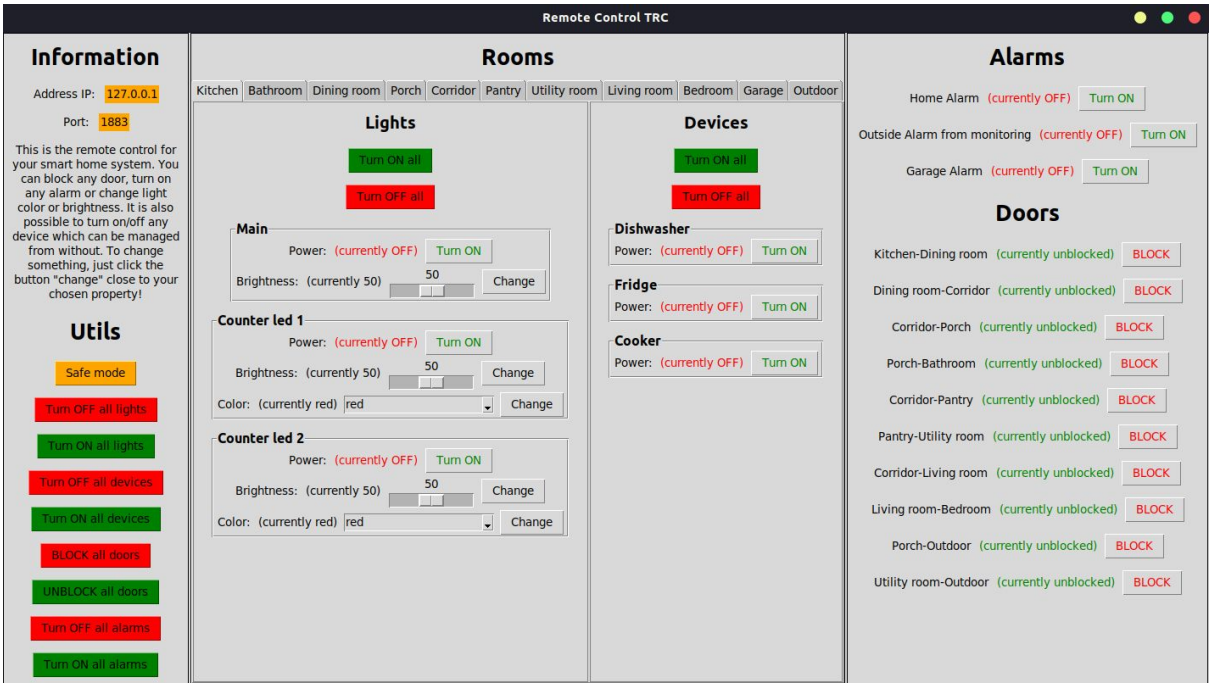
Smart Home Remote Control - Dokumentacja

Wstęp

Smart Home Remote Control to aplikacja służąca jako pilot inteligentnego domu. Umożliwia połączenie się poprzez sieć do naszego systemu i wydawanie poleceń, które mają np. uaktywnić alarm bądź wyłączyć światło w pokoju. Można to robić lokalnie tzn. łącząc się do serwera poprzez router w domu podając lokalne IP serwera i port lub z zewnątrz przez internet podając zewnętrzne publiczne IP serwera i port. Pilot nie narzuca nam schematu naszego systemu - jeżeli chcemy wykorzystać go do połączenia się z gotowym systemem smart, wystarczy zmienić konfigurację plików JSON tak aby zgadzała się ona z tym co posiadamy w naszym domu lub mieszkaniu. To wszystko nadaje to ogromnej elastyczności i wygody w użytkowaniu.

Instrukcja

Do uruchomienia wystarczy mieć zainstalowanego pythona oraz mosquitto (MQTT). Po pobraniu całej aplikacji, w głównym katalogu należy wykonać w konsoli polecenie “python main.py”. Uruchomi to okienko logowania, w którym trzeba wpisać odpowiedni adres IP oraz port serwera. Następnie pojawi się duże okienko pilota, z którego można wykonywać odpowiednie polecenia.



1. Zdjęcie przedstawia wygląd pilota

Aplikacja jest podzielona na kilka sekcji: (od lewej) informacje o połączeniu oraz przyciski funkcyjne (np. blokada wszystkich drzwi lub uaktywnienie wszystkich alarmów). Tuż obok znajduje się sekcja pokojów, w której możemy wybrać konkretny pokój a w nim włączyć/wyłączyć konkretne światła lub urządzenia. Dodatkowo, można sterować jasnością światła a także zmieniać jego kolor o ile pozwala na to samo oświetlenie w domu (określa się to na poziomie konfiguracji w plikach JSON. Dla każdego pokoju znajdują się również przyciski funkcyjne umożliwiające wykonanie tych czynności dla wszystkich znajdujących się w danym pokoju urządzeń lub świateł. Ostatnia sekcja to alarmy i drzwi - w niej możemy aktywować/dezaktywować alarm lub zablokować/odblokować drzwi. Wszystko co chcemy wykonać z poziomu pilota odbywa się za pomocą przycisków - to one ostatecznie wysyłają polecenie do serwera. Jeżeli chcemy zmienić kolor światła, należy najpierw wybrać odpowiedni kolor, następnie wcisnąć przycisk “change”. Analogiczna sytuacja występuje w przypadku jasności. Jeżeli chodzi o zasilanie - odpowiednio klikając przycisk turn on / turn off włączymy lub wyłączymy urządzenie/światło. To samo tyczy się alarmów oraz drzwi. Inaczej jednak działa to w przypadku przycisków funkcyjnych - ich kliknięcie powoduje wysłanie do serwera szeregu poleceń zmieniających stan obiektów na ten, wynikający z przycisku np. “block all doors” wyśle polecenie zablokowania drzwi do wszystkich tych które nie są jeszcze zablokowane.

Technologie

Wykorzystane technologie w tej aplikacji to:

- python - język w którym został napisany pilot,
- tkinter - graficzny interfejs aplikacji,
- MQTT - protokół transmisji danych, oparty o wzorzec publish/subscribe. W tym przypadku jest wykorzystywany do komunikacji z systemem inteligentnego domu - pilot publikuje polecenia w danym temacie i docierają one do serwera, który z kolei rozsyła je do wszystkich subskrybujących klientów,
- JSON - z plików tego typu wczytywana jest konfiguracja pilota,
- wzorzec projektowy Observer - jest pośrednikiem pomiędzy silnikiem aplikacji a jej graficznym interfejsem.

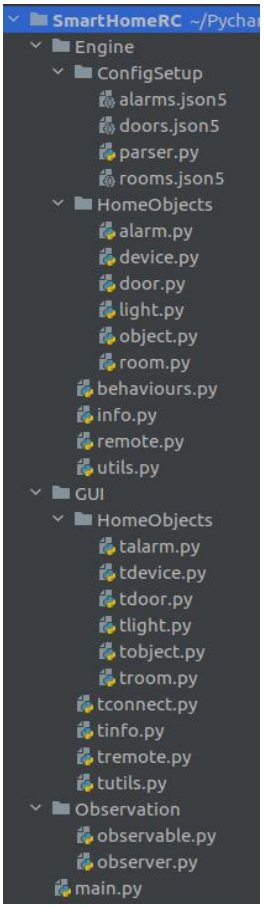
Budowa

Cechą charakterystyczną budowy tego projektu jest struktura katalogowa. W katalogu głównym znajdują się plik główny main.py który odpowiada za uruchomienie pilota oraz trzy katalogi:

- Engine - stanowi silnik aplikacji,
- GUI - graficzny interfejs,
- Observation - pośrednik między silnikiem a interfejsem

Najważniejszym katalogiem jest Engine. Bez niego aplikacja nie ma szans zadziałać. Bez graficznego interfejsu i pośrednika w postaci wzorca Observer aplikację można uruchomić a korzystanie z niej można zaimplementować w postaci integracji z użytkownikiem poprzez

konsolę. Sam silnik składa się na katalog odpowiedzialny za ładowanie konfiguracji pilota - są to pliki formatu JSON, które prezentują schemat oraz plik parser.py, który konwertuje obiekty formatu JSON na obiekty typu takiego jak te zdefiniowane w katalogu HomeObjects. Ten katalog z kolei zawiera definicje różnych klas na których pracuje aplikacja. Klasa Object w pliku object.py stanowi korzeń innych klas - Device, Light, Alarm dziedziczą po klasie Object, przy czym Light dodatkowo rozszerza ją, gdyż w przeciwieństwie do pozostałych, możemy zmieniać jasność oraz kolor światła. Door i Room są klasami niezwiązanymi z klasą Object ze względu na różnorodność danych. Pozostałe pliki z katalogu silnika stanowią istotną rolę w funkcjonalności pilota. Najmniejsze znaczenie odgrywa klasa Info - informuje ona tylko o połączeniu i pokazuje krótką instrukcję obsługi. Klasa Utils to funkcje, które upraszczają nam korzystanie z pilota (np. wyłącz wszystkie światła). Klasa Remote to rdzeń aplikacji - odpowiada za połączenie z serwerem, uruchomienie pilota oraz wymianę poleceń z serwerem. Definicje zachowań w przypadku odbioru polecenia, znajdują się w klasie Behaviours. Tak skonstruowany silnik jest warunkiem koniecznym działania pilota i wystarczającym w przypadku gdy mamy napisaną również integrację użytkownika za pomocą konsoli. Wiadomym, że jednak typowym użytkownikiem nie będzie programista tylko również osoba niedoświadczona w informatyce, zatem rozwiązanie konsolowe nie jest najlepszym rozwiązaniem. Na pomoc przychodzi tutaj folder GUI - odpowiada on za wyświetlanie i wykonywanie funkcjonalności silnika. Katalog HomeObjects zawiera tkinter'owe wersje klas z referencjami na obiekty klas z HomeObjects w Engine. Tak samo jak w silniku, najmniej przydatną klasą jest TInfo, która wyświetla to co zawiera obiekt klasy Info z silnika. TConnectWindow (plik TConnect) to klasa która stanowi okno łączenia się z serwerem. Jest to pierwsze okno jakie pojawia nam się po uruchomieniu pilota. Tam wpisujemy adres IP oraz port serwera, a po potwierdzeniu tego przyciskiem "start", pojawia nam się okno pilota, którego implementacja zawarta jest w klasie TRemote. Funkcjonalność pochodząca z silnika (Utils) jest w interfejsie zdefiniowana jako klasa TUtils. Zarówno w tym jak i innym przypadku, pliki katalogu GUI muszą korzystać z plików silnika, co jednak jeżeli wartości obiektów z silnika dynamicznie zmieniają się a nie widac tego w interfejsie? Rozwiązaniem tego problemu jest właśnie wzorzec projektowy Observer zawarty w katalogu Observation. Obiektami typu Observable są obiekty utworzony na podstawie klas silnika, zaś obiektami typu Observer są obiekty utworzone na podstawie klas graficznego interfejsu. W przypadku kiedy zmieni się jakaś wartość na obiektach z silnika, wszyscy obserwatorzy się o tym dowiedzą (w tym przypadku obiekty interfejsu) i na moment kiedy otrzymują to powiadomienie, zmieniają dane na panelu pilota. Dzięki zastosowaniu tego wzorca, uzyskałem nie tylko sprawną aktualizację danych, ale również rozłączność silnika aplikacji oraz silnika graficznego. Jest to szczególnie dobre rozwiązanie w przypadku, kiedy chcielibyśmy dokonać poważnej modyfikacji aplikacji - gdyby silnik został napisany tak aby współpracował z silnikiem graficznym to wtedy zmiana graficznego interfejsu nie byłaby możliwa bez zmian w silniku pilota.



Rozwój

Dzięki budowie, która została opisana powyżej, aplikacja jest gotowa na zmiany. Można swobodnie napisać GUI w innej technologii niż tkinter lub zastosować inne wzorce. Jedynym wyjątkiem jest niestosowanie wzorca Observer - w tym przypadku należy pamiętać, że jeżeli chcemy go usunąć to musimy zlikwidować dziedziczenie po Observable w plikach Engine/HomeObjects. Warto pamiętać, że może się to przydać zwłaszcza w przypadku kiedy GUI wspierałoby sygnały i sloty tak jak jest to np. w PyQt. Tutaj nie jest nam potrzebny wzorzec Observer, gdyż wszystko można zaimplementować na sygnałach i slotach. Gdybyśmy jednak nie chcieli modyfikować apki, a rozszerzyć ją - jest to również możliwe dzięki odpowiedniej budowie. Jeżeli chcielibyśmy dodać jakąś funkcjonalność do urządzeń w pokojach, wystarczy dodać ten parametr w plikach JSON (choć nie zawsze jest to konieczne), uwzględnić go w parserze, dodać odpowiedni atrybut w klasie Device, dodać mu zachowanie w przypadku odebrania polecenia (klasa Behaviours), a następnie zająć się widocznością tego atrybutu na panelu pilota (dodać Frame w TDevice a do niego odpowiednie struktury). Tak samo jeżeli chcielibyśmy dodać jakąś nietypową funkcjonalność, która miałaby ułatwiać nam obsługę pilota - wystarczy zdefiniować funkcję w klasie Utils i dodać przycisk w klasie TUtils obsługujący tę funkcję. Podsumowując, aplikacja jest napisana nie tylko tak, by umożliwiać modyfikację na własne potrzeby, ale również tak aby umożliwiać jej dalszy rozwój.

Autor:

Łukasz Wolski