Projekt z Programowania systemów internetu rzeczy - projekt SmartHome

Zespół :
Michał Podgajny 311412
Wojciech Kondracki 310941
Jędrzej Joniec 310880

Prowadzący dr inż. Jarosław Domaszewicz, mgr inż. Aleksander Pruszkowski

Politechnika Warszawska. Wydział Elektroniki i Technik Informacyjnych.

15 stycznia 2023

Spis treści

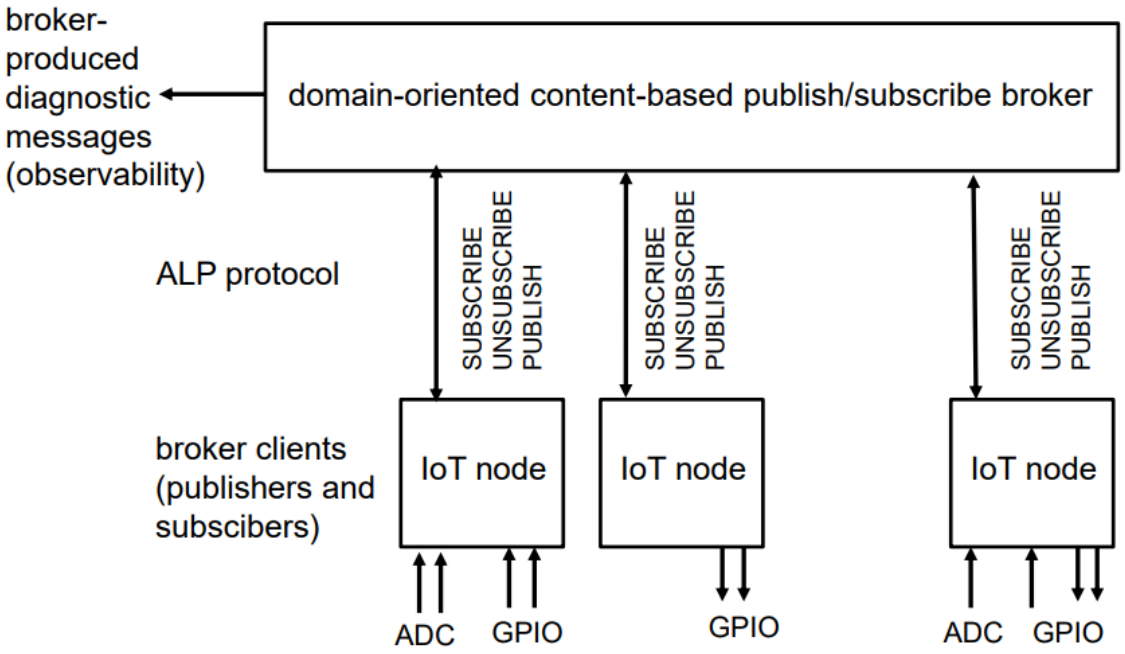
1. Wstęp

Naszym zadaniem w ramach projektu było wykonanie inteligentnego systemu zarządzania otoczeniem wraz z możliwością przesyłania danych pomiędzy urządzeniami w sieci (content-based) umożliwiający komunikację za pomocą protokołu *publish/subscribe*. W ramach projektu ustalono 3 główne instancje:

— **PUBLISHERS** - jest to ogół czujników wraz z dodatkową funkcjonalnością (np. czujnik jest zintegrowany z silnikiem podnoszącym lub opuszczającym rolety), które wysyłają informacje o swoim stanie (np. temperatura, jasność) jak i również mogą nasłuchiwać rozkazów pochodzących od klientów (np. aby zasunął roletę).
— **BROKER** - jest to główny serwer zarządzający siecią inteligentnego domu. Odpowiada on za to aby przekazać stan *publisherów* do klientów jak i umożliwić przekazanie rozkazu od *client*'a do *publishera*. Serwer takowy musi zapewnić izolację zarówno klientów jak i *publisherów* od warstwy fizycznej sieci (klient aby uzyskać informacje o temperaturze nie musi wiedzieć jaki ma numer czujnika odbiera temperaturę w danym pokoju, jedyne co mu potrzebne jest do uzyskania takowej informacji jest informacja intuicyjne powiązana z daną wartością mierzoną np. numer pokoju). *Broker* przekazuje jedynie wiadomości które interesują dany obiekt w sieci (*publisher*'ów lub klientów).
— **CLIENT** - jest to urządzenie które obsługują np. domownicy do zarządzania domem za pomocą dostępnych w sieci inteligentnego domu urządzeń (np. wcześniej wspominany moduł sterowania silnikiem zasuwającym rolety). Urządzenie to musi umożliwić odbieranie wiadomości przekazanych od *broker*'a oraz umożliwić ich interpretację.

W tym modelu czujniki przesyłają informację do *broker*'a a ten przekazuje ją do klienta. Podobnie też klient wysyłając rozkaz do danego urządzenia wysyła ten rozkaz do *broker*'a a ten następnie do odpowiedniego urządzenia, który może obsłużyć taki rozkaz. W tym celu używane są 3 typy wiadomości: **Subscribe** do wystosowania żądania wobec brokera aby przekazywał odpowiedniego typu informacje do strony zainteresowanej **Unsubscribe** do wystosowania żądania rezygnacji z przekazywania danego typu informacji oraz samej wiadomości **Publish** służącej do przekazywania właściwych informacji.

Celem tego projektu było stworzenie protokołu umożliwiającego realizację powyższej koncepcji, tzw *ALP* który będzie służył do przekazywania wiadomości pomiędzy urządzeniami, w taki sposób aby było to ergonomiczne sieci lokalnej urządzeń *IoT* czyli urządzeń o małych zasobach. Poniżej znajduje się schemat prezentujący działanie systemu:



2. Opis domeny - smart home

Wybraną przez zespół tematyką jest realizacja takowego systemu na potrzeby realizacji koncepcji *Smart Home*. Ludzie na ogół cenią sobie wygodę i upraszczanie codziennych czynności, w czym taki system może ich odciążyć oszczędzając ich cenny czas na pracę, naukę czy większą ilość czasu na odpoczynek. Ludzie na ogół są leniwi i wolą zamiast iść samemu zasunąć rolety czy zgasić żarówkę albo żeby jakiś system za nich to zrobił a w ostateczności żeby z poziomu np. aplikacji na telefon te aktywności zrealizować. Projekt ten ma pomóc w zautomatyzowanym zarządzaniu domem.

Na potrzeby realizacji tego systemu postanowiono zgłębić temat. Przeanalizowano więc potrzeby domowników. Duża ich część zależy od parametrów takich jak jasność (często domy są tak ulokowane, że o pewnych porach dnia jest w danym pokoju bardzo ciemno a w innych bardzo jasno, co utrudnia np. zdalną pracę, naukę albo po prostu zwykłe wygodne funkcjonowanie), temperatura (nikt nie chce żyć w zimnym domu, ale również gdy w domu jest zbyt ciepło nie jest to wygodne dla domowników), wilgotność (duża wilgotność w domu sprzyja np. pojawianiu się ognisk grzyba na ścianach oraz życie w permanentnie wilgotnym pomieszczeniu nie jest najlepsze dla ludzkiego zdrowia warto jest informować domowników o zbyt dużej wilgotności) oraz ciśnienia (jest to informacja szczególnie istotna dla starszych ludzi, którzy z powodu zmian ciśnienia odczuwają duży dyskomfort, warto jest ich o tym poinformować aby odróżnić ten dyskomfort od innych powodów które mogą wymagać pomocy lekarskiej).

W ramach tego projektu skupiono się głownie na zarządzaniu światłem. Gdy w danym pokoju jest zbyt jasno powinna być możliwość zdalnego zasunięcia rolet a jak zbyt ciemno to włączenia żarówek lub podniesienia rolet. W ten sposób można zaoszczędzić na kosztach energii nie marnując jej gdy jest dostępne naturalne światło. Poza tym głównym celem systemu jest to aby informować domowników o wcześniej wspomnianych istotnych parametrach.

2.1. Opis mierzonych wartości

Urządzenia związanie z tym systemem IoT mierzą 4 wartości:

— **Temperaturę** (*temperaturę*) - określoną w zakresach wartości 0-100 (założono że w mieszkaniu w którym żyją ludzie temperatura nie spadnie poniżej zera). Aby zamienić tą wartość na rzeczywistą temperaturę starczy uzyskaną wartość podzielić na 10. Przedstawiona wartość jest po prostu dziesięciokrotnością temperatury określonej w skali *Celsjusza*.
— **Ciśnienie** (*pressure*) - określoną w zakresie wartości 0-1023 gdzie wartość ta określa ciśnienie w *hektopaskalach* i nie wymaga żadnej dalszej zamiany.
— **Jasność** (*lightness*) - określona w skali 0-1023 gdzie 0 to najmniejsza jasność a 1023 to największa, gdzie jednak zwykłe dzienne światło w pochmurny dzień określa się w przedziale 300-400
— **Wilgotność** (*humitidy*) - określona w skali 0-100 która przedstawia wartość wilgotności w skali procentowej, ona podobnie jak ciśnienie nie wymaga zamiany.
— **Info** (*info*) - jest to dowolna wartość informacyjna przekazywana przez czujniki. Dla tego systemu przesyła ona stan baterii danego czujnika (o ile sam czujnik jest zasiany za pomocą baterii to np. system opuszczania rolet nie) w skali 0-1023 gdzie 0 to rozładowana bateria a 1023 to pełna. Wartość tą na potrzeby testów *zahardcoreowano* w kod węzła.

2.2. Opis struktury fizycznej

W ramach projektu zrealizowano wyżej wspomniany system dla małego mieszkania 2-pokojowego (salon + sypialnia). Urządzenia umieszczono w salonie oraz sypialni. Urządzenia wchodzące w skład sieci urządzeń IoT tego domu są emulowane za pomocą dostarczonego przez prowadzącego emulatora *Arduino*, które mają zaimplementowane wyjścia i wejścia cyfrowe oraz wejścia analogowe (ADC). Ze względu na specyfikę systemu nie użyto w tym projekcie wejść cyfrowych, używano jedynie wejść analogowych do odczytywania szukanych wartości. Urządzenia podano z numerami ID które będą potrzebne później do zestawienia komunikacji w brokerze.

2.2.1. Salon

Ponieważ analizujemy mieszkanie w tzw. nowym budownictwie oznacza to iż aneks kuchenny jest połączony z salonem i tworzą jeden pokój. Z tego powodu w tym pokoju musi być więcej czujników. Zainstalowano tutaj 3 czujniki

2.2.2. Węzeł 1 Czujnik temperatury (ID: 501)

Jest to prosty czujnik który przesyła temperaturę (czyta ją z modułu *ADC* oznaczonego sygnaturą **A3**) do brokera i jest związany z całym pokojem, gdyż pozostałe urządzenia w tym pokoju nie posiadają termometru. Urządzenie to nie nasłuchuje poleceń a jedynie cyklicznie co pewien ustalony w kodzie węzła czas wysyła informacje o temperaturze.

2.2.3. Węzeł 2 Urządzenie wielofunkcyjne przy rolecie jednego z okien(ID: 502)

Urządzenie to znajduje się przy oknie obok wyjścia na balkon. Jest ono zintegrowanym czujnikiem jasności (pobiera on napięcie z *ADC* oznaczonym symbolem **A4**) z modułem sterowania silnikiem rolet. Urządzenie to umożliwia zdalne podnoszenie i opuszczanie rolet. Urządzenie to zgodnie z oczekiwaniami nasłuchuje rozkazów związanych z włączeniem lub wyłączeniem silnika rolety oraz jego kierunku jego działania - czy ma zasuwać czy opuszczać rolety (stąd mamy dwa wyjścia *D2* - praca silnika o **D3** - kierunek pracy silnika). Urządzenie automatycznie wyłącza silnik po pewnym ustalonym czasie o ile ten nie zostanie wyłączony wcześniej. Urządzenie to będąc również czujnikiem wysyła cyklicznie (co czas ustalony w kodzie urządzenia)informacje o mierzonej wartości do brokera.

Urządzenie ulokowano tutaj gdyż jest to miejsce najbardziej wrażliwe na wpływ światła oraz jest blisko rolet, zatem opłacalnym jest zainstalować taki moduł blisko urządzenia związanego z daną aktywnością.

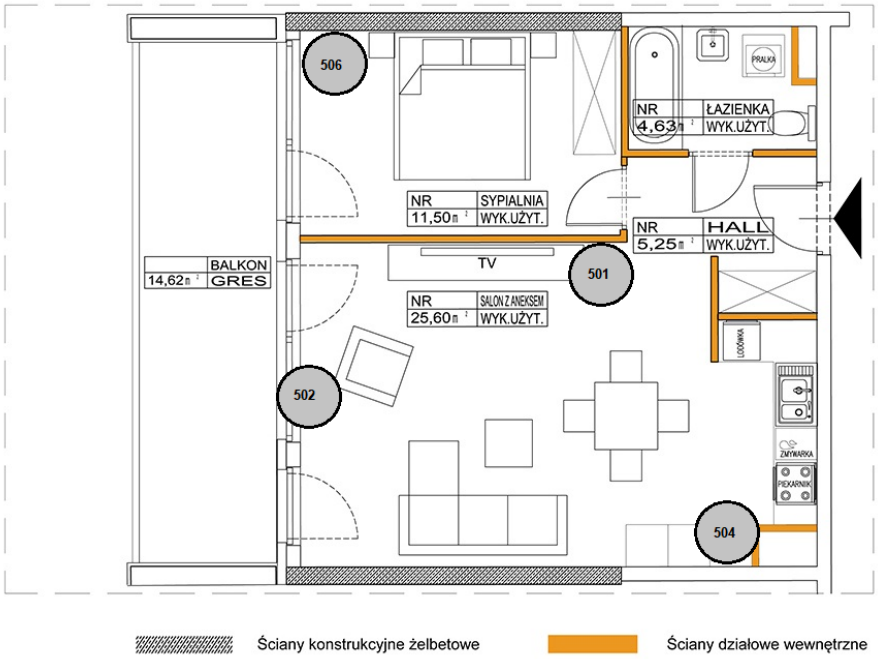
2.2.4. Węzeł 3 Inteligentna żarówka obok kuchenki (ID: 504)

Jest to urządzenie będące zarówno żarówką jak i czujnikiem 3 parametrów: wilgotności i ciśnienia. Urządzenie to dodatkowo nasłuchuje rozkazów związanych z gaszeniem lub zapalaniem żarówki. Wysyła ono cyklicznie, co pewien czas określony w kodzie urządzenia do brokera informacje o mierzonych wartościach. Urządzenie samodzielnie nie wykonuje żadnych aktywności (np. po pewnym czasie gaśnie żarówka). Żarówka ma taką budowę żeby temperatura samej żarówki nie wpływała na pomiar (na specjalnym spiczastym końcu).

2.3. Sypialnia

2.3.1. Węzeł 4 Inteligentna żarówka w sypialni (ID: 506)

Żarówka ta zachowuje się podobnie jak ta z salonu ale mierzy ona dodatkowo temperaturę.
Poniżej znajduje się schemat z naniesionymi na plan mieszkania czujnikami:



2.4. Opis struktury logicznej

Jednym z celów projektu który wyznaczył sobie zespół jest izolacja zarówno *publisherów* jak i klientów od warstwy fizycznej urządzeń *IoT*. Celem jest odwołanie się do danego kontekstu (np do temperatury w danym pokoju) a nie do danego czujnika. Istniała więc potrzeba stworzenia systemu, który umożliwiłby powstanie warstwy logicznej, która będzie bardziej intuicyjna z perspektywy domownika (nie będącego osobą techniczną). Do tego celu użyto drzewiastej struktury danych.

Umożliwia one bardzo łatwe odwołanie się do wszystkich elementów związanych z danym szerszym tematem (do danej gałęzi, przykładowo zebrać temperatury z całego mieszkania lub z całego pokoju nie musząc odwoływać się do każdego elementu struktury logicznej osobno). Do tej struktury dodano jednak specjalne zbiory na odnośniki (które będą dalej nazywane *shortcuts*) do węzłów do każdej gałęzi tworząc strukturę na kształt klasycznego systemu plików. Drzewo to dzięki skrótom umożliwia to iż do danego węzła można odwołać się na różne sposoby (np. chcemy sprawdzić temperaturę w pokoju Kasi (szukając w zbiorze osób) ale też możemy sprawdzić temperaturę w pokoju numer 12 (szukając w zbiorze pokojów)).

2.4.1. Struktura drzewa

Drzewo składa się z gałęzi (*branch*). Każda z takich gałęzi posiada zbiór skrótów związanych z daną gałęzią (może on być pusty). Dodatkowo każda z takich gałęzi posiada odniesienie do potencjalnej podgałęzi. W ten sposób można tworzyć teoretycznie nieskończoną (ograniczoną pamięcią brokera) strukturę w której można powiązać logicznie różne mierzone wielkości (*quantities*) oraz aktywności (*activities*) związanych z danym tematem w oderwaniu od fizycznej struktury urządzeń. Skróty są związane z węzłami (w warstwie logicznej).

2.4.2. Podzbiory węzłów logicznych

Węzły logiczne można podzielić na 2 kategorie

— mający swojego przedstawiciela w warstwie logicznej - są to zazwyczaj urządzenia małej mocy z których klient może pobierać mierzone dane
— nie mające takiego przedstawiciela (roboczo nazwano taki węzeł logiczny *driver*) - stanowią one skrót dla klienta, dzięki któremu klient w modelu komunikacji *publish-subscribe* może komunikować się z danym węzłem (nie wiedząc o jego istnieniu) w celu wykonania pewnej aktywności (np. zgaszenia żarówki).

2.4.3. Drzewo opisanej powyżej sieci

Broker w celu stworzenia warstwy logicznej sieci urządzeń *IoT* używa pliku konfiguracyjnego `tree.txt` który zawiera konfigurację takowego drzewa. Poniżej znajduje się jego treść:

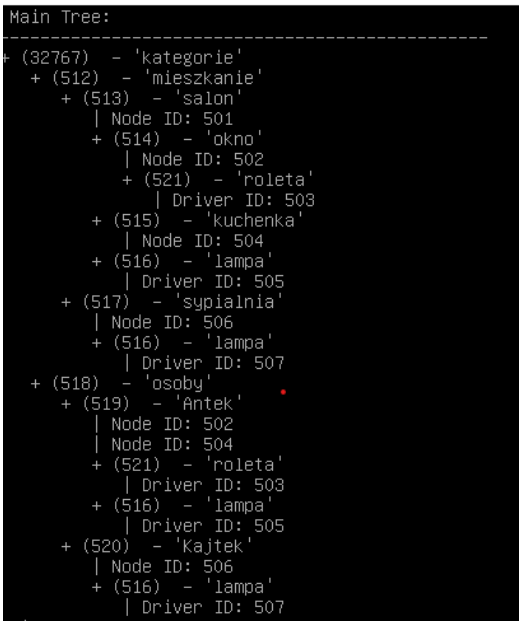
```
/mieszkanie/salon=501
/mieszkanie/salon/okno=502
/mieszkanie/salon/okno/roleta+503
/mieszkanie/salon/kuchenka=504
/mieszkanie/salon/lampa+505
/mieszkanie/sypialnia=506
/mieszkanie/sypialnia/lampa+507
/osoby/Antek=502
/osoby/Antek=504
/osoby/Antek/roleta+503
/osoby/Antek/lampa+505
/osoby/Kajtek=506
/osoby/Kajtek/lampa+507
```

Węzły logiczne przed których numerami stoi plus są to pseudowęzły będące skrótami do sterowania, zaś węzły oznaczone znakiem = są to węzły mające swoich przedstawicieli w warstwie fizycznej. Widać tutaj połączenia z opisanymi wcześniej węzłami (gdyż numery ich zgadzają się z ujętymi w ścieżkach drzewa kategoriami).

Utworzono dwie kategorie dzięki którym można się odwoływać do żądanych wartości lub sterowania daną aktywnością: *mieszkanie* gdzie podzielono mieszkanie względem pokojów oraz miejsc, gdzie dane aktywności można wykonać lub jakąś wielkość można zmierzyć oraz *osoby* gdzie można odwołać się do danych wielkości związanych z daną osobą. Na podstawie tego możemy zauważyć np. że Antek mieszka normalnie w salonie a Kajtek w sypialni. Mamy więc dwa konteksty dzięki którym możemy odwoływać się do wielkości związanych z interesującym nas tematem skupiania się na szukaniu fizycznego numeru węzła.

Numery zaimplementowano gdyż nie chciano ustalać na sztywno IP węzła, stworzono więc specjalny zakres adresów dla węzłów. Adres taki jest przesyłany przez węzeł w wiadomości *HELLO* w celu zestawienia węzła z uzyskanym *IP*. Aby system działał poprawnie numery muszą być unikatowe.

Poniżej znajduje się screen wygenerowanego schematu drzewa na podstawie powyższego pliku konfiguracyjnego:



3. Mechanizm subskrypcji

W celu zgłoszenia potrzeby otrzymania żądanej informacji zaistniała potrzeba powstania mechanizmu subskrypcji. Umożliwia on wysłanie za pomocą odpowiednich wyrażeń logicznych itp. informacji na temat interesujących daną stronę tematów (pomiarów itd.)

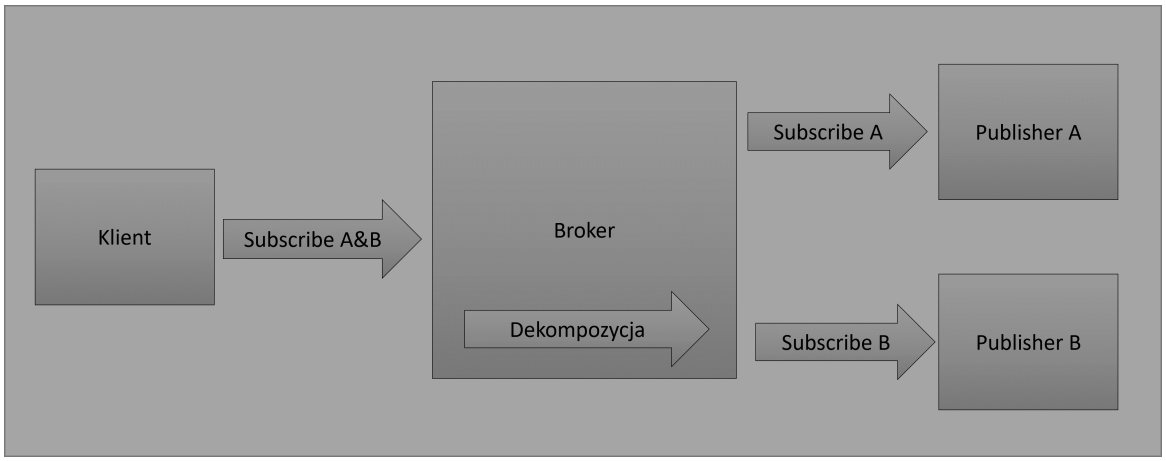
3.1. Kto i po co używa subskrypcji w tym systemie

W tym systemie subskrypcji używają

— **Klient** - po to aby zgłosić potrzebę uzyskania danych informacji uzyskanych za pomocą czujników, bez wiedzy o ich fizycznym istnieniu. Klient dodatkowo odwołuje subskrypcje za pomocą wiadomości *UNSUBSCRIBE*. Klienci wysyłają zazwyczaj złożone wiadomości tego typu uwzględniające jedynie węzły logiczne uwzględniające rzeczywiste czujniki (istnieje ograniczenie w systemie na wybieranie węzłów niefizycznych w niższych warstwach danej gałęzi tak aby nie można było wybrać węzłów logicznych będącymi skrótami, które nie znajdują się bezpośrednio w danej gałęzi a nie w podgałęziach).

— **Publisher** - po to aby nasłuchiwać rozkazów od klienta. Wysyła on jedynie na początku swojego działania wiadomość *SUBSCRIBE* i z reguły nie odwołuje subskrypcji. Wysyła on krótką wiadomość związaną jednie z danym skrótem do którego będą odwoływać się klienci.
Dzięki takiemu zestawieniu zarówno *publisher*'zy jak i klienci mogą uzyskać potrzebne im informacje.

Schemat działania wiadomości subscribe znajduje się poniżej.



3.2. Schemat tworzenia wiadomości

Wiadomość tworzona jest w następujący sposób Najpierw określa się ścieżkę do danego miejsca w drzewie. Ta ścieżka jest zapisywana w oszczędniejszej formie (na podstawie dołączonego słownika w pliku konfiguracyjnym *dict.txt*). Ścieżki te łączone są za pomocą wplatanych między nimi wyrażeń logicznych (suma logiczna *OR* lub iloczyn logiczny *AND*) tworząc zapytanie (roboczo nazwane *query*). Następnie do takiego wyrażenia na koniec dołącza się *wildcard*'e dzięki której można określić jakie usługi mają zostać wybrane (jakie wartości mierzone, czy ma być nasłuchiwany rozkaz, czy zbierać wszystkie dane z danej gałęzi itp). Na koniec na sam początek wiadomości jest dokładany nagłówek świadczący o tym że jest to wiadomość *SUBSCRIBE*. Analogicznie tworzona jest wiadomość *UNSUBSCRIBE*. Schemat tak powstałej wiadomości znajduje się poniżej:

```
1  subscribe # identyfikator
2  <shorten_name_branch> # rzad kolejnych skrconych nazw galezi tworzy scieke.
3  <shorten_name_branch>
4  <shorten_name_branch>
5  <logic_expression> # wyraenie logiczne spajajace scieki
6  <shorten_name_branch>
7  <shorten_name_branch>
8  <shorten_name_branch>
```

```
    <shorten_name_branch>
    <logic_expression>
    ...
    <shorten_name_branch>
    <logic_expression>
    <shorten_name_branch>
    <shorten_name_branch>
    <shorten_name_branch>
    <wildcard> # określenie adanych uslug
```

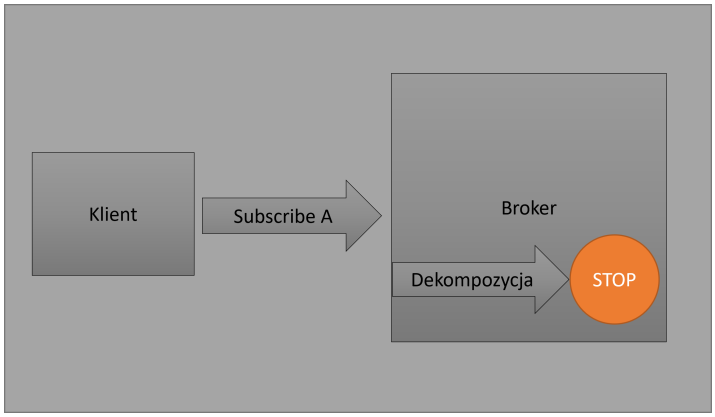
3.3. Dekompozycja zapytania subskrypcyjnego na mniejsze

Wiadomość po dotarciu do brokera jest stosowanie dekomponowana (zarówno dla publikacji jak i rezygnacji z publikacji). Szuka się wszystkich węzłów znajdujących się w gałęzi o określonej w wiadomości ścieżce. Po wczytaniu pierwszych dwóch zbiorów (o ile wcześniej nie wystąpiła sekcja *wildcardy*) jest na nich wykonywana zgodnie z zapamiętanym separatorem operacja iloczynu lub sumy logicznej. Zbiór takowy jest zapisywany a następnie po wyczytaniu węzłów z kolejnej gałęzi wykonywana jest operacja logiczna między wcześniej zapamiętanym zbiorem a uzyskanym z ścieżki zbiorem.

Proces ten kończy się aż zostanie wykryta sekcja *wildcardy*, kończąca wiadomość. W ten sposób uzyskiwane są też ścieżki z którymi dalej będą przesyłane wiadomości *PUBLISH* które są zapamiętywane na poczet dalszego przekazywania wiadomości. Przykład efektów takiej dekompozycji widać poniżej (warto odnieść się do wcześniej wspomnianego drzewa):

```

[2023-01-17 00:40:13] MESSAGE: ALP Subscribe Message. SOURCE IP ***
Selecting single nodes to forwadring ALP Publish messages from this nodes:

 New subscriber IP: ***


 Operations :
PATH: osoby/Kajtek
OR PATH: osoby/Antek

 Results:
Filters : All services
-------------------------------------------------
* NODE ID : 506 in path: /osoby/Kajtek
* NODE ID : 502 in path: /osoby/Antek
* NODE ID : 504 in path: /osoby/Antek
-------------------------------------------------
```

Następnie do uzyskanych w ten sposób węzłów wysyła się wiadomo wiadomość *SUBSCRIBE* z zdekomponowanymi ścieżkami, aby poinformować węzły że są subskrybowane przez kogoś (nie uzyskują informacji przez kogo). Wiadomość taka nie jest przekazywana dalej w przypadku węzłów logicznych które nie mają swojego fizycznego odpowiednika a są jedynie skrótem dla klienta do wysyłania rozkazów. Mechanizm przetwarzania wiadomości dla takich węzłów znajduje się poniżej:

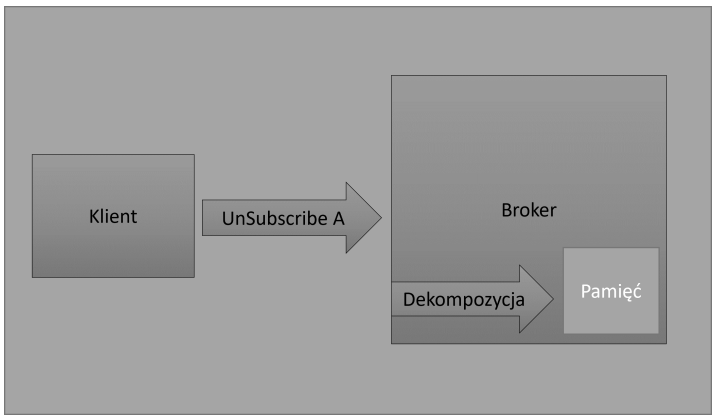


3.4. Usuwanie subskrypcji

Usuwanie subskrypcji zamiast dodawać wyznaczone w powyższy sposób węzły z listy subskrypcji danego subskrybenta usuwa je. Szczególnym przypadkiem jest wysłanie wiadomości *UNSUBSCRIBE* z ścieżką o wartości `0x0 0x1`, co powoduje usunięcie subskrybenta z listy subskrybentów. Schemat wiadomości *UNSUBSCRIBE* wygląda następująco:

```
1   unsubscribe # identyfikator
2   <shorten_name_branch> # rzad kolejnych skrconych nazw gal ezi tworzy ściek e.
3   <shorten_name_branch>
4   <shorten_name_branch>
5   <logic_expression> # wyraenie logiczne spajaj ace ścieki
6   <shorten_name_branch>
7   <shorten_name_branch>
8   <shorten_name_branch>
9   <shorten_name_branch>
10  <logic_expression>
11  ...
12  <shorten_name_branch>
13  <logic_expression>
14  <shorten_name_branch>
15  <shorten_name_branch>
16  <shorten_name_branch>
17  <wildcard> # określenie adanych uslug
```

Informacja o rezygnacji z subskrypcji nie jest wysyłana dalej. Jes to spowodowane tym, że wiadomość *PUBLISH* traktuje się jako *trigger*, mający wywoływać odesłanie jakiejś wartości na życzenie a nie czekając na stan. Schemat rezygnacji z subskrypcji zaprezentowano poniżej:



4. Mechanizm publikacji informacji

Aby można było przekazywać konkretne informację pomiędzy urządzeniami w sieci urządzeń *IoT* należało stworzyć specjalną wiadomość służącą tylko i wyłącznie do tych celów.

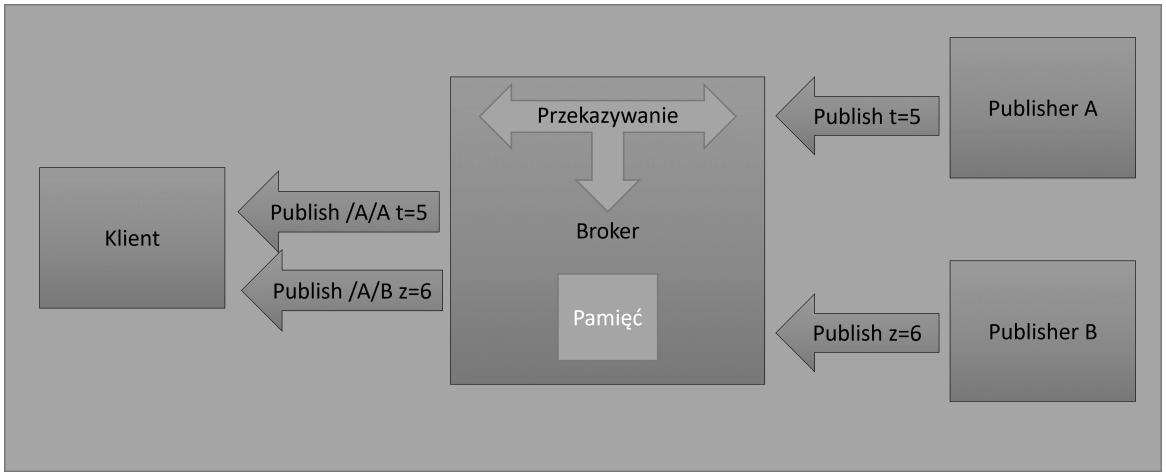
4.1. Kto i po co używa mechanizmu publikacji informacji

W tym systemie publikacji używają

— **Klient** - po to aby wydać rozkaz który umożliwi wykonanie odpowiedniej akcji (np. zgaszenie światła). Będą to wiadomości krótkie, gdzie nie licząc ścieżki oraz nagłówka będzie się składać jedynie z jednego bajta. Wiadomość wysyłana przez klienta będzie miała jednak pełną ścieżkę tak aby można było ją przekierować do celu po czym po przejściu przez *broker* jest upraszczana, aby węzeł nie musiał przetwarzać długiej wiadomości.

— **Publisher** - po to aby wysłać do brokera informację, która będzie dalej przekazywana do zainteresowanych subskrybentów. Wiadomość jaką wysyła publisher ma uproszczoną postać (gdyż węzeł ma małe zasoby) aby potem dopisać do niego ścieżkę nadając kontekst wiadomości, która potem jest odsyłana zgodnie z listą subskrypcyjną do danych subskrybentów.

Schemat takiego połączenia znajduje się poniżej:



4.2. Schemat tworzenia wiadomości

Istnieją dwie formy wiadomości *PUBLISH*:

— **Uproszczona** - wysyłana przez *publisherów*.

Wiadomość ta tworzona jest kolejno: najpierw dodawany jest nagłówek wiadomości *PUBLISH*. Następnie dodawana jest fałszywa ścieżka, która umożliwia odróżnienie wiadomości uproszczonej od tej z nadanym kontekstem. Potem określa się rodzaj wysłanej usługi za pomocą *wildcardy* takiej samej jak w przypadku wiadomości *SUBSCRIBE*. Następnie określa się długość wiadomości. Wiadomości w *PUBLISH* są podzielone na 3 klasy:

— **Rozkazy** o długości jednego bajta służąca do przekazywania rozkazów

— **Wartości sensorów** o długości dwóch bajtów, służąca do przekazywania wartości z czujników

— **Metadane** o długości 3 i więcej bajtów, służąca do przekazywania metadanych, które potrafią mieć dużą ilość bajtów.

. Następnie zgodnie z ustaloną wartością pobierana jest informacja. Schemat takiej wiadomości znajduje się poniżej:

```
1  publish # identyfikator
2  <shorten_name_branch> # tym razem ścieka jest prawdziwa
3  <shorten_name_branch>
4  ...
```

```
    <shorten_name_branch>
    <shorten_name_branch>
    length_of_value # na podstawie jej określa si e długość dalszej wiadomości.
    <value>
    <value>
    ...
    <value>
```

— **Z nadanym kontekstem** - przetworzona przez brokera z nadanym nagłówkiem uwzględniający kontekst (gdyż wiadomość pochodząca z węzła nie ma takich informacji, aby uprościć przetwarzanie w węźle).

Początek tworzenia takiej wiadomości jest podobny jak dla wiadomości *SUBSCRIBE*. Tworzona jest ścieżka ale następnie od razu jest dodawana tzw. *wildcard*-a. Dodawany jest nagłówek oraz treść wiadomości na koniec podobnie jak dla wiadomości uproszczonej. Schemat takiej wiadomości wygląda następująco:

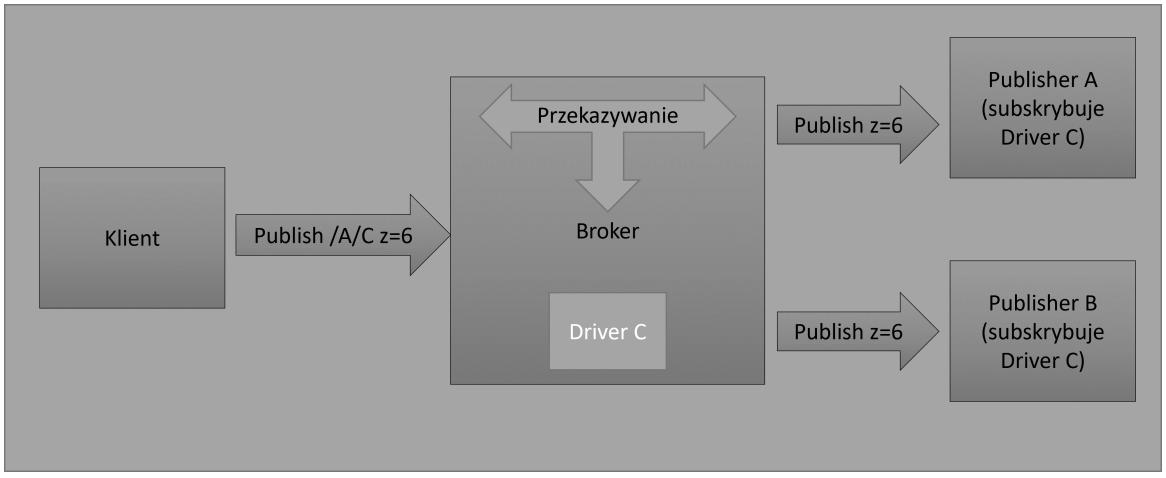
```
    publish # identyfikator
    <fake_path> # Falszywa ścieka
    <wildcard>
    length_of_value # na podstawie jej określa si e długość dalszej wiadomości.
    <value>
    <value>
    ...
    <value>
```

4.3. Dalsze przekazywanie wiadomości do odbiorców

Na podstawie informacji zapisanych podczas dekompozycji wiadomości *SUBSCRIBE* możliwe jest dalsze przekazywanie wiadomości. Na podstawie listy subskrypcji określane są *IP* subskrybentów oraz ścieżki które pozwalają określić kontekst wiadomości. Do wiadomości dodawana jest wcześniej wspominana ścieżka po czym jest przekazywana do odbiorców (klientów).

Z goła odmienna sytuacja jest w przypadku wysyłania rozkazu poprzez skrót. Do *broker*'a dociera wiadomość wraz z informacjami o jej kontekście (ścieżka drzewa). Podczas przetwarzania jej ta ścieżka jest usuwana aby uprościć wiadomość, żeby węzeł nie musiał przetwarzać dużej ilości bajtów. Tak przetworzona wiadomość jest wysyłana do *publisherów* który subskrybują skrót.

Schemat wysyłania rozkazu znajduje się poniżej:



5. Specyfikacja protokołu ALP

Aby zrealizować wcześniej omawiany protokół należało przetworzyć go tak aby jak najmniej bajtów zajmował. Powstała potrzeba utworzenia protokołu warstwy aplikacji *ALP* w postaci bitowej co zostanie opisane poniżej:

5.1. Typy wiadomości

Zgodnie z wcześniejszym omówieniem konceptu protokołu protokół ma 4 typy wiadomości

— **HELLO** o identyfikatorze `0x4` służąca do rejestracji węzłów o małych zasobach do systemu.
— **PUBLISH** o identyfikatorze `0x3` służąca do przesyłania informacji oraz rozkazów.
— **SUBSCRIBE** o identyfikatorze `0x1` służąca do wysyłania próśb o subskrybcje
— **UNSUBSCRIBE** o identyfikatorze `0x2` służąca do rezygnacji z subskrypcji

Nagłówek w tym protokole składa się z 1 bajtu. Pierwsze 5 bitów zajęte są przez identyfikator aplikacji (dla tego protokołu będzie to `0x15`). Identyfikator ten służy do tego aby odróżnić wiadomość pochodzącą od innej aplikacji która omyłkowo przyszła na port tejże aplikacji od właściwej wiadomości protokołu. Następne 3 bajty określają typ wiadomości.

Opis niniejszych identyfikatorów znajduje się w pliku `broker&subscriber/ALP/packets.c`.

```
1  #define SUBSCRIBE 0x1
2  #define UNSUBSCRIBE 0x2
3  #define PUBLISH 0x3
4
5  #define HELLO 0x4
```

Z części wiadomości tam omówionych zrezygnowano wraz z rozwojem projektu w celu uproszczenia. Pierwotnie typ wiadomości miał zajmować 3 bity obecnie mógłby zajmować 2, zrezygnowano z tego rozwiązania z myślą o dalszym rozwoju tego protokołu (zostawiono 3 bajty). Budowę nagłówka przedstawiono na poniższym schemacie:

8	7	6	5	4	3	2	1
I	I	I	I	I	T	T	T

Gdzie `I` to identyfikator aplikacji a `T` to typ wiadomości.

5.2. Tworzenie ścieżki oraz zapytania logicznego

Podstawowym elementem z którego składa się duża część wiadomości w niniejszym protokole. Ścieżka zgodnie z poprzednimi ustaleniami składa się z uproszczonych nazw gałęzi. Uproszczenie to polega na stworzeniu słownika który zamienia nazwy na liczby w zakresie 0-2^{15}. W ten sposób zamiast długiego ciągu znaków jedna nazwa w ścieżce zawiera się w 2 bajtach. Dzięki zastosowaniu słownika rośnie wyspecjalizowanie tego systemu, gdyż ścieżka może składać się jedynie z ściśle określonych słów. Rozmiar słownika dobrano tak aby można było tworzyć sieci węzłów o *de-facto* nieograniczonym rozmiarze, przy okazji wysyłając stosunkowo małe wiadomości.

Uzyskana liczba po przetworzeniu przez słownik jest przetwarzana do formy odpowiedniej dla tego protokołu (tzn. przesunięta w lewo o 1). Ostatni bit zostawiono aby na podstawie niego określać czy ścieżka się zakończyła. Gdy ścieżka się kończy ostatni bit jest równy 1. Wygląda to w następujący sposób:

8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
L15	L14	L13	L12	L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1	0

A gdy zakodowana nazwa jest ostatnią nazwą w ścieżce:

8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
L15	L14	L13	L12	L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1	1

Gdzie *Ln* to kolejne bity liczby uzyskanej z słownika

Dzięki takiemu rozwiązaniu ścieżka nie ma ograniczenia długości sama w sobie (Ogranicza ją jedynie bufor analizujący ścieżki o długości `128` bajtów). Z takich ścieżek potem mogą składać się wyrażenia logiczne. Aby dodać wyrażenie logiczne po wykryciu ostatniego bajtu ścieżki, dodawany jest separator. Może on pełnić funkcje zarówno `wildcardy` jak i oznaczenia operacji logicznej. Budowa takiego separatora wygląda następująco

8	7	6	5	4	3	2	1
T	T	0	S	S	S	S	E

Gdzie:
— Bity **T** oznaczają typ separatora. Istnieją 3 typy separatora:
  — `OR_OPERATION` gdzie bity oznaczone literą T przyjmują wartość `0x1` - oznacza to operację sumy logicznej z następnym zbiorem określonym ścieżką.
  — `AND_OPERATION` gdzie bity oznaczone literą T przyjmują wartość `0x2` - oznacza to operację iloczynu logicznego z następnym zbiorem określonym ścieżką.
  — `WILDCARD` gdzie bity oznaczone literą T przyjmują wartość `0x3` - określa to iż jest to koniec wyrażenia logicznego po którym znajduje się *wildcard*'a określająca to jakie usługi ma obejmować opisywany zbiór. Usługi te określone są w bitach S
— Bity **S** oznaczają typ usługi, uwzględnianej podczas analizowania tzw. *wildcardy*. Istnieje 6 takich usług:
  — `ALL_SERVICES` o wartości `0x0` określa że trzeba uwzględnić wszystkie usługi.
  — `TEMPERATURE` o wartości `0x1`
  — `PRESSURE` o wartości `0x2`
  — `HUMITIDY` o wartości `0x3`
  — `LIGHTNESS` o wartości `0x4` to wartości odczytywane przez czujniki.
  — `COMMAND` o wartości `0x5` oznacza to iż trzeba uwzględnić miejsce w którym można przesyłać rozkazy.
  — `INFO` o wartości `0x6` służy do uwzględnienia węzłów które wysyłają wiadomości diagnostycznej (np. stan baterii).

  Typy usług określono w plikach `brokersubscriber/ALP/packets/wildcard.c`:

```
1  #define ALL_SERVICES 0x0
2  #define TEMPERATURE 0x1
3  #define PRESSURE 0x2
4  #define HUMITIDY 0x3
5  #define LIGHTNESS 0x4
6  #define COMMAND 0x5
7  #define INFO 0x6
```

— Bit **E** czyli bit końca wyrażenia logicznego. Gdy jego wartość wynosi `0x1` należy przestać analizować takowe wyrażenie.

Wyrażenie logiczne powstałe w ten sposób może przybrać taką postać:

8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
L15	L14	L13	L12	L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1	0
8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
L15	L14	L13	L12	L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1	1
8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
T	T	0	S	S	S	S	0	L15	L14	L13	L12	L11	L10	L9	L8
8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
L7	L6	L5	L4	L3	L2	L1	1	T	T	0	S	S	S	S	1

Dzięki tym parametrom możliwe jest sprecyzowanie kontekstu dla poleceń. Do takiego wyrażenia dokładany jest na początku nagłówek i w zależności od typu wiadomości dokładane są bajty do końca pakietu. W tej postaci łatwo jest też wykryć koniec zapytania znajdując w 2 bajtach pod rząd na końcu analizowanego bajtu(operacja `varriable & 0x1`) jedynki.

5.3. Wiadomość PUBLISH

Wiadomość *PUBLISH* składa się z nagłówka powstałego zgodnie ze specyfikacją (pierwsze 5 bitów identyfikatora aplikacji a pozostałe 3 bity identyfikator wiadomości - dla wiadomości `PUBLISH` jest to `0x3`). Następnie znajduje się wyrażenie logiczne. Po jego zakończeniu w kolejnym bajcie znajduje się długość danych zawartych w sektorze danych tego pakietu (nazwijmy ten bajt bajtem długości wiadomości). Następnie wiadomość składa się z tylu bajtów ile określono w bajcie długości wiadomości.

Długość wiadomości zawartej w pakiecie wiadomości *PUBLISH* ograniczona jest przez wielkość bajtu określającego długość wiadomości (maksymalnie 255 bajtów - takie wartości jednak nie są raczej osiągalne a wiadomości używane w `IoT` są krótsze jednak zostawiono taką furtkę dla nietypowych sytuacji).

Schemat wiadomości znajduje się poniżej. Ilość bajtów wiadomości (60) jest obrazowa może być każda w przedziale `1-255`, zgodnie z ilością określoną za pomocą bitów *L*;

8	7	6	5	4	3	2	1	...	...	6	5	4	3	2	1
1	0	1	0	1	0	1	1	...	...	0	S	S	S	S	1

8	7	6	5	4	3	2	1	8	...	6	5	4	3	2	1
L	L	L	L	L	L	L	L	D1	...	D55	D56	D57	D58	D59	D60

Wiadomość w formie uproszczonej zamiast bitów ścieżki posiada bajty kolejno `0x0` i `0x1`, po czym znajduje się *wildcard*'a informująca o przesyłanym rodzaju informacji.

5.4. Wiadomość SUBSCRIBE

Aby utworzyć wiadomość PUBLISH do utworzonej wcześniej ścieżki należy dodać na sam początek jedynie nagłówek o stosownej wartości. Nagłówek zgodnie ze specyfikacją składa się z identyfikatora (`0x15`) znajdującego się w pierwszych 5 bajtach (binarnie ta liczba prezentuje się następująco `10101`) oraz identyfikatora wiadomości w ostatnich 3 bitach (`0x1`). Stworzona w ten sposób wartość ma następującą postać:

8	7	6	5	4	3	2	1	...	...	6	5	4	3	2	1
1	0	1	0	1	0	0	1	...	...	0	S	S	S	S	1

Wiadomość kończy się wraz z końcem z końcem wyrażenia logicznego wraz z wildcardą i to na podstawie tego obliczana jest długość wiadomości do przesyłu.

5.5. Wiadomość UNSUBSCRIBE

Podobnie jak w poprzednim przypadku do utworzonej wcześniej ścieżki należy dodać na sam początek jedynie nagłówek o stosownej wartości. Jeżeli chodzi o budowę binarną wiadomość ta różni się od poprzedniej jedynie identyfikatorem typu wiadomości (`0x2`). Stworzona w ten sposób wartość ma następującą postać:

8	7	6	5	4	3	2	1	...	...	6	5	4	3	2	1
1	0	1	0	1	0	1	0	...	...	0	S	S	S	S	1

Wiadomość kończy się wraz z końcem z końcem wyrażenia logicznego wraz z wildcardą i to na podstawie tego obliczana jest długość wiadomości do przesyłu.

5.6. Wiadomość HELLO

Wiadomość *HELLO* jako jedyna nie korzysta z ścieżki w swojej treści, gdyż tego nie wymaga. Za pomocą niej węzeł o małych zasobach informuje broker o tym jakie wartości mierzy itp. Schemat wiadomości znajduje się poniżej:

8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
1	0	1	0	1	1	0	0	N	N	N	N	N	N	N	N

8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
N	N	N	N	N	N	N	N	0	0	0	0	0	0	0	0

8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1
T	P	H	L	C	I	0	1	-	-	-	-	-	-	-	-

Wiadomość przystosowano do większej ilości usług ale ze względu na brakujący czas zrealizowano tylko 6 usług. Wiadomość składa się z nagłówka z identyfikatorem aplikacji i identyfikatorem wiadomości. Następne dwa bajty określają numer węzła, za pomocą którego broker będzie identyfikował go i na podstawie którego będzie on przesyłał mu dane.Kolejne dwa bity określają dostępne przez meldujący się w brokerze węzeł usługi (w ten sposób zestawia się połączenie). Bity określone literami określają możliwość do publikacji danych usług:

— *I* - info
— *C* - rozkaz
— *L* - poziom oświetlenia
— *H* - wilgotność
— *P* - ciśnienie
— *T* - temperatura

Wiadomość kończy się bitem końca (`0x1`). Jest to jedyna wiadomość o stałej długości (wynosi ona 5 bajtów).

6. Implementacja serwera

6.1. Reprezentacja obiektów rzeczywistych w serwerze

Aby zrealizować niniejszą koncepcje należało utworzyć reprezentacje obiektów w brokerze. Potrzebne było stworzenie zasadniczo 3 instancji:

— **Drzewa** składająca się z struktur typu `tree` której deklaracja zawarta jest w pliku `branch.c`

```
1       typedef struct tree{
2           short id;
3           struct tree* next_branch;
4           struct tree* subbranch;
5           shortcut* nodes;
6       } tree;
```

Dzięki takiej budowie mogą one tworzyć strukturę drzewiastą gdzie `subbranch` jest odniesieniem do gałęzi "odrastającej" od tego odcinka gałęzi a `next_branch` jest jakby przedłużeniem tej gałęzi.Dzięki odniesieniu do listy węzłów logicznych, może on tworzyć strukturę katalogową. Dodatkowo każda gałąź posiada swój numer za pomocą którego przy użyciu słownika można odkodować nazwę danej gałęzi. Struktura `shortcut` jest bardzo podobna do struktury która będzie opisana później a mianowicie `lnode` i służy ona do tworzenia list węzłów logicznych (postanowiono jednak wydzielić obiekt na potrzeby drzewa, aby zaoszczędzić dane):

```
1       typedef struct shortcut{
2           node* node;
3           struct shortcut* next_short;
4
5       } shortcut;
```

Deklaracja tej struktury jest w pliku `shortcuts.c`

— **Listy subskrybentów**. Lista ta znajduje się w pliku `subscribers.c` i składa się z struktur `subscriber` o niniejszej deklaracji:

```
1       typedef struct subscriber{
2           int ip;
3           char connected;
4
5           serviceNode* services;
6
7           struct subscriber* next_sub;
8       }subscriber;
```

Zawiera ona odpowiednio przetworzone adres IP subskrybenta, odnośnik do następnego subskrybenta dzięki czemu można stworzyć listę oraz odnośnik do listy usług subskrybowanych przez danego klienta. Lista usług składa się z struktur

```
typedef struct serviceNode{
    node* myNode;
    char* path;
    char typeSensor;

    struct serviceNode* next_item;
}serviceNode;
```

które zawierają poza odnośnikiem do kolejnej usługi ścieżkę oraz typ usługi (sensora) ustaloną na podstawie procesu dekompozycji (opis tej struktury znajduje się w pliku `services.c`). Struktura ta zawiera też odniesienie do samego węzła o której liście będzie potem

— **Listy węzłów**. Składa się ona z specjalnych struktur `lnode`, które są odnośnikami do informacji na temat węzłów zawartych w strukturach `node`. Deklaracja tych strutów znajduje się poniżej oraz w pliku `nodes.c`:

```
typedef struct node
{
    int numberNode;
    int ip;

    char driver;

    char temperature;
    char pressure;
    char humitidy;
    char lightness;
    char command;
    char info;

    char connected;
}node;

typedef struct lnode
{
    node* myNode;
    char* path;
    int ipSubscriber;

    struct lnode* next_item;
}lnode;
```

Struktura informacyjna zawiera informacje o *IP* węzła (o ile jest to węzeł fizyczny), numerze (`numberNode`) oraz na temat dostępnych czujników oraz usług. Dodatkowo dostępna jest informacja o tym czy węzeł jest tylko skrótem czy ma fizyczny odpowiednik (parametr `driver` - gdy wynosi 1 to oznacza że jest to jedynie skrót a jak 0 oznacza to że jest odnośnikiem do prawdziwego węzła).

Struktura `lnode` zawiera więcej szczegółów niż `shortcut` i wynika to z faktu, iż ta struktura jest używana podczas wykonywania dekompozycji polecenia *PUBLISH* na niej wykonywane są operacje logiczne itp. oraz jest jedną z struktur tworzące listy wybranych węzłów za pomocą funkcji. Podobne też wykorzystywana jest struktura `serviceNode` gdy wykonywane są operacje na listach węzłów uwzględniających typ usługi z jakimi są związane.

6.2. Mechanizm subskrypcji

Mechanizm subskrypcji w szczegółach działa następująco:
— Wiadomość jest odbierana przez serwer oraz rozpoznawana jako wiadomość `SUBSCRIBE`.
— Analizowane jest wyrażenie logiczne za pomocą funkcji `analyseQuery()`

```
1
2  lnode* analyseQuery(char* query){
3
4      lnode* finalResult = NULL;
5  ...
6
7      for(n=0;n<QUERY_LEN;n+=2){
8          if(n+1 < QUERY_LEN){
9
10             pathBuffer[k] = query[n];
11             pathBuffer[k+1] = query[n+1];
12
13             if(query[n+1] & 0x1){
14
15                 if(n+2 < QUERY_LEN){
16
17 ...
18
19                       switch (lastOperation){
20                          case NO_OPERATION:
21
22                              printf("PATH:␣");
23                              printShortPath(pathBuffer);
24                              printf("\n");
25
26                              finalResult = selectAll(pathBuffer);
27                              break;
28                          case OR_OPERATION:
29
30                              printf("OR␣PATH:␣");
31                              printShortPath(pathBuffer);
32                              printf("\n");
33
34                              set = selectAll(pathBuffer);
35                              finalResult = orItem(finalResult,set);
36
37
38                              break;
39  ...
40 }
```

Jak widać po wyodrębnieniu konkretnych ścieżek wraz z uzyskaniem ostatnio wykrytego separatora opisującego operację logiczną odczytuje się za pomocą operacji `selectAll` wszystkie węzły związane z daną ścieżką na drzewie. Po takim wyodrębnieniu następnie są wykonywane operacje logiczne aż do wykrycia *wildcardy*; Po dalszym przetwarzaniu zamiany ścieżki na konkretną gałąź funkcja selectAllNodes() wybiera wszystkie węzły związane z danym kontekstem za pomocą poniższej funkcji:

```
1      static void addAllNodesDirectlyLocated(tree* analysedBranch,char* path, int level){
2
3      char* relPath = malloc(sizeof(char) * DIRECOTRY_BUFFER_SIZE);
4      copyPath(relPath,path);
5
```

```
6       if(analysedBranch !=NULL){
7           shortcut* analysedNode = analysedBranch -> nodes;
8
9           while (analysedNode != NULL)
10          {
11              if(analysedNode -> node != NULL){
12
13                  copyPath(relPath,path);
14
15                  char* newPath = malloc(sizeof(char) * DIRECOTRY_BUFFER_SIZE);
16                  copyPath(newPath,relPath);
17
18                  if(!analysedNode -> node -> driver || level == 1){
19                      addSelectedItem(analysedNode -> node,newPath,0);
20                  }
21
22
23              }
24              analysedNode = analysedNode -> next_short;
25          }
26      }
27
28      free(relPath);
29  }
30
31  static void selectAllNodesR(tree* branch,char* path,char addBranchToP, int level){
32
33      level ++;
34      char* relPath = malloc(sizeof(char) * DIRECOTRY_BUFFER_SIZE);
35      memset(relPath,0,sizeof(char) * DIRECOTRY_BUFFER_SIZE);
36
37      copyPath(relPath,path);
38
39      tree* analysedBranch = branch;
40
41
42      while (analysedBranch != NULL)
43      {
44          copyPath(relPath,path);
45          if(addBranchToP) addBranchToPath(relPath,analysedBranch -> id);
46
47          addAllNodesDirectlyLocated(analysedBranch,relPath,level);
48          if(analysedBranch -> subbranch != NULL) selectAllNodesR(analysedBranch -> subbranch,relPath,TRUE,level);
49          analysedBranch = analysedBranch -> next_branch;
50      }
51
52      free(relPath);
53
54
55  }
56
57  void selectAllNodes(tree* branch,char* absPath){
58      initNewSelectionItems();
59      char* relPath = malloc(sizeof(char) * DIRECOTRY_BUFFER_SIZE);
60      memset(relPath,0,sizeof(char) * DIRECOTRY_BUFFER_SIZE);
61
62      copyPath(relPath,absPath);
63
64      int level = 0;
65
66      selectAllNodesR(branch,relPath,FALSE,level);
```

```
67      free(relPath);
68  }
```

Dzięki uwzględnieniu poziomu zagnieżdżenia nie można wybrać nie odwołując się bezpośrednio do węzła logicznego będącego skrótem. Idąc po kolei po drzewie wybierane są kolejne węzły aż do uzbierania wszystkich potrzebnych.
Następnie wartości są filtrowane na podstawie wildcardy za pomocą tejże funkcji znajdującej się w pliku `wildcard.c`

```
1
2   lnode* sensorFilter(lnode* set, char service){
3
4       results = NULL;
5
6       lnode* analysedItem = set;
7       char detected = FALSE;
8
9       while(1){
10
11          if(analysedItem -> myNode != NULL){
12
13              node* myNode = analysedItem -> myNode;
14              char decision = FALSE;
15
16              switch(service){
17                  case ALL_SERVICES:
18                    decision = TRUE;
19                    break;
20                  case TEMPERATURE:
21                    if(myNode ->temperature) decision = TRUE;
22                    break;
23      ...
```

Po utworzeniu listy węzłów związanych z zapytaniem lista takich usług jest dodawana i zwracana

```
1       ...
2       lastService = service;
3
4       if(!detected) return NULL;
5       else{
6
7
8         lnode* prefinal = sensorFilter(orset,service);
9
10        showSelectedItems(prefinal);
11        addAllToListSensorNode(prefinal, service);
12
13        lastOpeationSelectedNodes = copySelectedItems(prefinal);
14
15        killCopiedList(prefinal);
16        printf("\n␣\n␣Subscribed␣services␣by␣");
17        printf("%s",decodeIPtoCharacters(ip));
18        printf("\n");
19
20        return duplicateServicesList(serviceSet);
21      }
```

Lista taka jest zapisywania przy danym subskrybencie w jego strukturze podpinając się do listy usług `serviceNode`. Następnie przesyłane są dalej wiadomości o subskrypcji:

```
1
2  void forwardSubscrieMessage(char service){
3      lnode* analysedNodes = lastOpeationSelectedNodes;
4
5      if(analysedNodes != NULL)
6        while(1){
7
8          if(analysedNodes -> myNode != NULL){
9
10             if(analysedNodes -> myNode -> ip != 0){
11                 show_time();
12                 printf("Sending␣to␣IP:␣%s␣SUBSCRIBE␣Message␣\n", decodeIPtoCharacters(analysedNodes -> myNode -> ip
13
14                 char* spath = analysedNodes -> path;
15                 char* query = newQuery(spath);
16
17                 query = addWildcard(query,service);
18
19                 char* myMessage = makeSubscribeMessage(query);
20
21                 free(spath);
22
23                 int len = packetLen(myMessage);
24
25                 send_it(myMessage,decodeIPtoCharacters(analysedNodes -> myNode -> ip), len);
26
27                 countSSubscribeMessages++;
28                 free(myMessage);
29             }
30         }
31
32         if(analysedNodes -> next_item == NULL) break;
33         analysedNodes = analysedNodes -> next_item;
34     }
35 }
```

6.3. Mechanizm przekazywania publikowanych wiadomości dalej

W przypadku wysyłania wiadomości `PUBLISH` przez węzeł odczytywany jest adres *IP* hosta wysyłającego daną wiadomość. Następnie odczytywany jest z tego numer ID węzła. Następnie flirtuje się wszystkie subskrypcje pod kątem danego *IP* i typu usług

```
1      ...
2        printf("]\n");
3
4      node* node = getNodeByIp(ip);
5
6      if(node != NULL){
7          printf("Publish␣message␣is␣associated␣with␣*␣NODE␣%d␣\n",node -> numberNode);
8
9          lnode* selected = sendAboutNode(node -> numberNode,service);
10         showSelectedItems(selected );
11         printf("\n");
12         printf("Sending␣messages␣PUBLISH:␣\n");
```

```
        printf("\n");

        forwardPublishMessage(selected,content,service,len);
        killCopiedList(selected);
    }else{
        printf("ALERT! Service from this node is not subscribed by any subscriber \n");
    }

    free(content);
```

Na podstawie w ten sposób uzyskanej listy można dalej przekazywać wiadomości `PUBLISH`. Finalnie wiadomości rozsyłane są za pomocą tej funkcji

```

void forwardPublishMessage(lnode* nodes,char* message,unsigned char service, int len){
    lnode* analysedNode = nodes;
    if(analysedNode != NULL)
        while(1){

            if(analysedNode -> myNode != NULL){
                char* packetMessage = initPublishMessage();

                int n = 0;

                char* nodePath = analysedNode->path;

                while(1){
                    packetMessage[n+1] = nodePath[n];

                    if(packetMessage[n+1] & 0x1) break;
                    n++;
                }
                n++;
                packetMessage[n + 1] = ((WILDCARD << 6) | (service << 1) | 0x1)&0xFF;
                n++;
                packetMessage[n + 1] = len;
                n++;

                for(int k=0;k<len;k++){
                    packetMessage[n + 1] = message[k];
                    n++;
                }

                int len = n+1;

                show_time();
                printf("Sending to IP: %s PUBLISH Message \n", decodeIPtoCharacters(analysedNode -> ipSubscriber));

                send_it(packetMessage,decodeIPtoCharacters(analysedNode -> ipSubscriber), len);

                free(packetMessage);
                countSPublishMessages++;
            }

            if(analysedNode -> next_item == NULL) break;
            analysedNode = analysedNode -> next_item;
        }

}
```

7. Opis implementacji węzłów

7.1. Odczytywanie wiadomości od subskrybenta

Węzeł nasłuchuje wiadomości. Po wykryciu wiadomości, rozmiar pakietu jest większy od zera więc program przechodzi dalej. Po odczytaniu pakietu i gdy po jego przetworzeniu (enkapsulacja), rozmiar nadal jest większy od zera przechodzimy od odbierania wiadomości.

W naszej implementacji węzła *publisher* odbiera zasadniczo dwa typy wiadomości **PUBLISH** czyli rozkazy sterujące od subskrybenta oraz **SUBSCRIBE** czyli przekazane dalej po subskrybowaniu kontekstu w którym znajdował się dany węzeł. Po rozróżnieniu tego typu wiadomości mikrokontroler przechodzi do analizy.

```
1     void loop() {
2
3   int packetSize=Udp.parsePacket();
4   if(packetSize>0){
5
6       int len=Udp.read(packetBuffer, PACKET_BUFFER_LENGTH);
7
8       if(len > 0){
9           switch(get_message_type(packetBuffer[0])){
10              case SUBSCRIBE:
11                  subscribeAnalyse(len);
12                  break;
13              case PUBLISH:
14                  analysePublish(len);
15                  break;
16              default:
17                  break;
18          }
19
20      }
21
22  }
23    ...
```

Analizę działania zacznijmy od analizy działania po odebraniu wiadomości **SUBSCRIBE**. Zgodnie z oczekiwaniami jest wywoływany swojego rodzaju `trigger` i jest przesyłana zahardcodeowana zgodnie z wcześniejszymi założeniami stan baterii jako wiadomość określona usługą `info`. Dodatkowo zapisywana jest informacja o aktywacji węzła.

```
1
2
3   void subscribeAnalyse(int len){
4       Serial.println(F("I␣am␣subscribed␣by␣broker␣Service␣:␣"));
5
6       unsigned char service = (packetBuffer[len - 1] >> 1)& 0xF;
7
8       if(service == INFO_ID || service == ALL_SERVICES_ID){
9           publishSensor(INFO_ID,512);
10      }
11      active = TRUE;
12  }
```

Bardziej skomplikowaną analizę przeprowadza się dla wiadomości PUBLISH. Sprawdzane jest to czy wysłano rozkaz. Po ustaleniu tego sprawdzana jest czy uwzględniony rozkaz istnieje w wykazie. Jeżeli tak to wykonywana jest dana aktywność (np. włączany jest pin imitujący światło żarówki itd.).

```
1
2
3  void analysePublish(int len){
4      unsigned char service = (packetBuffer[3] >> 1)& 0xF;
5
6      if(service == COMMAND_ID){
7          Serial.println(F("Driver␣publish␣message:␣"));
8
9          Serial.println(service);
10
11         if(packetBuffer[4] == 1){
12             Serial.println(F("Command:␣"));
13              unsigned char command = packetBuffer[5];
14
15              switch(command){
16                 case TURN_ON_LIGHT_ACTIVITY:
17                     if(LIGHT){
18                          setPinStatus(LIGHT_PIN, HIGH);
19                         Serial.println(F("Light␣ON"));
20                     }
21                     break;
22                 case TURN_OFF_LIGHT_ACTIVITY:
23                     if(LIGHT) {
24
25                          setPinStatus(LIGHT_PIN, LOW);
26                          Serial.println(F("Light␣OFF"));
27                     }
28                     break;
29                 case TURN_ON_ENGINE_OF_BLINDS:
30                     if(BLINDS){
31
32                          setPinStatus(POWER_ENGINE_PIN, HIGH);
33                         timeTurnOnEngine = ZsutMillis();
34                          Serial.println(F("Binds␣engine␣ON"));
35                     }
36
37                     break;
38                 case TURN_OFF_ENGINE_OF_BLINDS:
39                     if(BLINDS){
40
41                          setPinStatus(POWER_ENGINE_PIN, LOW);
42                         timeTurnOnEngine = 0;
43                          Serial.println(F("Blinds␣engine␣OFF"));
44                     }
45                     break;
46                 case BLINDS_UP:
47                     if(BLINDS) {
48                          setPinStatus(DIRECTION_PIN, HIGH);
49                          Serial.println(F("Up␣Blinds"));
50                     }
51                     break;
52                 case BLINDS_DOWN:
53                     if(BLINDS){
54
55                          setPinStatus(DIRECTION_PIN, LOW);
56                          Serial.println(F("Down␣Blinds"));
57                     }
58                     break;
59             }
```

```
60
61        }
62    }
63
64 }
```

7.2. Wysyłanie publikowanych wiadomości

Kluczową funkcją wszystkich węzłów jest to aby cyklicznie wysyłać swój stan. Węzły mają określony za pomocą *define* parametr `TIMEOUT_SENSOR` pokazującą ile należy czekać na to aby wysłać następny pakiet wiadomości. Następnie po przekroczeniu tego czasu zgodnie z deklaracjami w *defineach* wysyłane są stosowne parametry dalej. Niektóre z parametrów wymagają aktywacji za pomocą wiadomości `SUBSCRIBE` tj. wilgotność itd.

```
1
2  void updateByTime(void){
3    if(ZsutMillis() - lastTime >= TIMEOUT_SENSOR){
4
5        Serial.println("Send␣sensor␣data␣to␣broker");
6
7        if(HUMITIDY){
8            if(active) {
9                publishSensor(HUMITIDY_ID,ZsutAnalog1Read());
10               Serial.println(F("+␣Humitidy"));
11           }
12       }
13
14       if(PRESSURE){
15           if(active){
16                publishSensor(PRESSURE_ID,ZsutAnalog2Read());
17                Serial.println(F("+␣Pressure"));
18            }
19       }
20       if(TEMPERATURE){
21           publishSensor(TEMPERATURE_ID,ZsutAnalog3Read());
22            Serial.println(F("+␣Temperature"));
23       }
24       if(LIGHTNESS){
25           publishSensor(LIGHTNESS_ID,ZsutAnalog4Read());
26            Serial.println(F("+␣Lightness"));
27       }
28
29       Serial.println(ZsutMillis());
30       lastTime = ZsutMillis();
31   }
32
33 }
```

W celu skrócenia kodu i oszczędności pamięci zamiast pisać w każdym przypadku od nowa generowanie pakietu użyto jednej funkcji opisanej poniżej, umożliwiającą łatwe wygenerowanie pakietu *PUBLISH*

```
1     void publishSensor(char service, short value){
2     packetBuffer[0] = code_caption(PUBLISH);
3     packetBuffer[1] = 0x0;
4     packetBuffer[2] = 0x1;
5     packetBuffer[3] = (WILDCARD << 6) | (service << 1) | 0x1;
```

```
    packetBuffer[4] = 0x2;

    packetBuffer[5] = (value >> 8)& 0xFF;
    packetBuffer[6] = (value) & 0xFF;

    response(7);
}
```

7.3. Subskrybowanie przez węzeł w celu umożliwienia sterowania nim

Węzeł aby mógł nasłuchiwać rozkazów pochodzących do subskrybentów musi wysłać do brokera wiadomość **SUBSCRIBE** aby broker przekierowywał wiadomości będące rozkazami do analizowanego węzła. Poniższa funkcja umożliwia to na podstawie uproszczonej postaci ścieżki drzewa:

```

#define BUDYNEK 512
#define MIESZKANIE2 514
#define POKOJ2 516
#define SZAFA 517

...

short pathCommand[] = {BUDYNEK,MIESZKANIE2,POKOJ2,SZAFA};
```

Dane te pobrano ze słownika i wybrano tylko taką ilość jaka jest potrzebna węzłowi do wygenerowania odpowiedniej ścieżki. Następnie można było taką wiadomość `SUBSCRIBE` związaną z kontekstem opisanym w ścieżce można było wysłać za pomocą poniższej funkcji.

```


void subscribeCommand(short* path,int len){
       packetBuffer[0] = code_caption(SUBSCRIBE);
       int n = 0;
       for(n=0;n<len;n++){
           packetBuffer[2*n + 1] = (path[n] >> 7) & 0xFF;
           packetBuffer[2*n + 2] = (path[n]<<1) & 0xFF;
       }
       packetBuffer[2*(n-1) + 2] = packetBuffer[2*(n-1) + 2] | 0x1;
       packetBuffer[2*(n-1) + 3] = (WILDCARD << 6) | (COMMAND_ID << 1) | 0x1;

       response(2*(n-1) + 4);

}
```

8. Przykład użycia

W celu testowania niniejszego protokołu opracowano 2 proste programy testowe wysyłające jedną wiadomość do brokera z poziomu klienta po czy sprawdzały one co się dalej działo i czy efekt był zgody z oczekiwaniami.

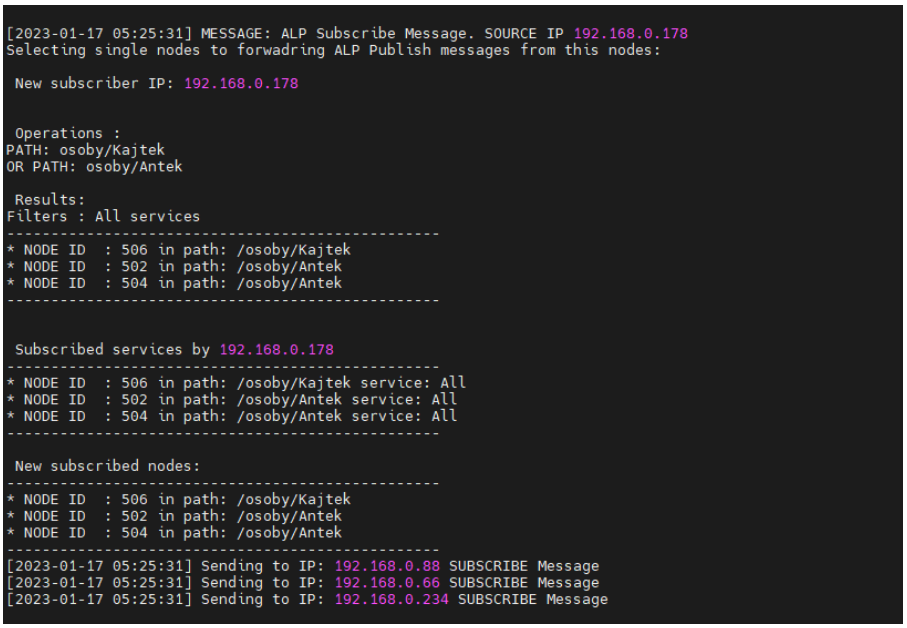
Najpierw zestawiono połączenie. Gdy wszystkie węzły wysłały wiadomość *HELLO* rozpoczęto realizację zadań.

8.1. Odczyt wartości z czujników

Pierwszym zadaniem było odczytanie danych z czujników związanych z osobami zamieszkującymi to mieszkanie. Do tego celu wykorzystano zapytanie z wyrażeniem logicznym (suma logiczna), aby sprawdzić działanie tej funkcjonalności. W tym celu stworzono prosty program który wysyła wiadomość i odbiera wiadomości które odeśle mu *broker*. Do generowania wiadomości wykorzystano cały pakiet modułów w c tak aby tworzenie tej wiadomości było prostsze i sprowadzało się do tego:

```
    void test1(){
    int len = 0;
    // Test2
    // Message2
    char* path1 = "/osoby/Kajtek";
    char* spath1 = encodePath(path1);
    char* query1 = newQuery(spath1);

    path1 = "/osoby/Antek";
    spath1 = encodePath(path1);
    query1 = addPathToQuery(query1,spath1,OR_OPERATION);

    query1 = addWildcard(query1,ALL_SERVICES);

    char* message1 = makeSubscribeMessage(query1);

    free(spath1);

    len = packetLen(message1);

    sendMessage(message1,len);
    free(message1);
}
```

Wiadomość sformułowano w taki sposób aby broker przesyłał wszystkie czujniki związane z Antkiem i Kajtkiem. Po kompilacji programu oraz jego uruchomieniu serwer pokazał dekompozycje wiadomości *SUBSCRIBE*. Uzyskano poniższy efekt:



Efekt jest zgodny z oczekiwaniami. Dekompozycja rozbiła wiadomość `SUBSCRIBE` na 3 mniejsze które są bezpośrednimi połączeniami z węzłami. Są to węzły numer *502,504* oraz *506*. Zgodnie z oczekiwaniami nie ma tutaj

węzła *501* (gdyż on nie został powiązany z żadną z wymienionych osób).
Ostateczną weryfikacją były wiadomości, które docierały do subskrybenta, potwierdzające poprawność powstałego w brokerze przekierowania:

```
[2023-01-17 05:25:39] MESSAGE ALP Publish Message  SOURCE IP: 192.168.0.178
Content Path: osoby/Antek
Service:+ pressure
Message:
  Type: Sensor value

1030



[2023-01-17 05:25:39] MESSAGE ALP Publish Message  SOURCE IP: 192.168.0.178
Content Path: osoby/Antek
Service:+ lightness
Message:
  Type: Sensor value

360



[2023-01-17 05:25:40] MESSAGE ALP Publish Message  SOURCE IP: 192.168.0.178
Content Path: osoby/Kajtek
Service:+ pressure
Message:
  Type: Sensor value

1020



[2023-01-17 05:25:40] MESSAGE ALP Publish Message  SOURCE IP: 192.168.0.178
Content Path: osoby/Kajtek
Service:+ temperature
Message:
  Type: Sensor value

210



[2023-01-17 05:25:40] MESSAGE ALP Publish Message  SOURCE IP: 192.168.0.178
Content Path: osoby/Kajtek
Service:+ lightness
Message:
  Type: Sensor value

360
```

Możliwa jest więc skuteczna subskrypcja i odbiór wiadomości. Dzięki temu mogliśmy odczytać szukane wartości, które są związane z danymi osobami (nie z danymi węzłami).

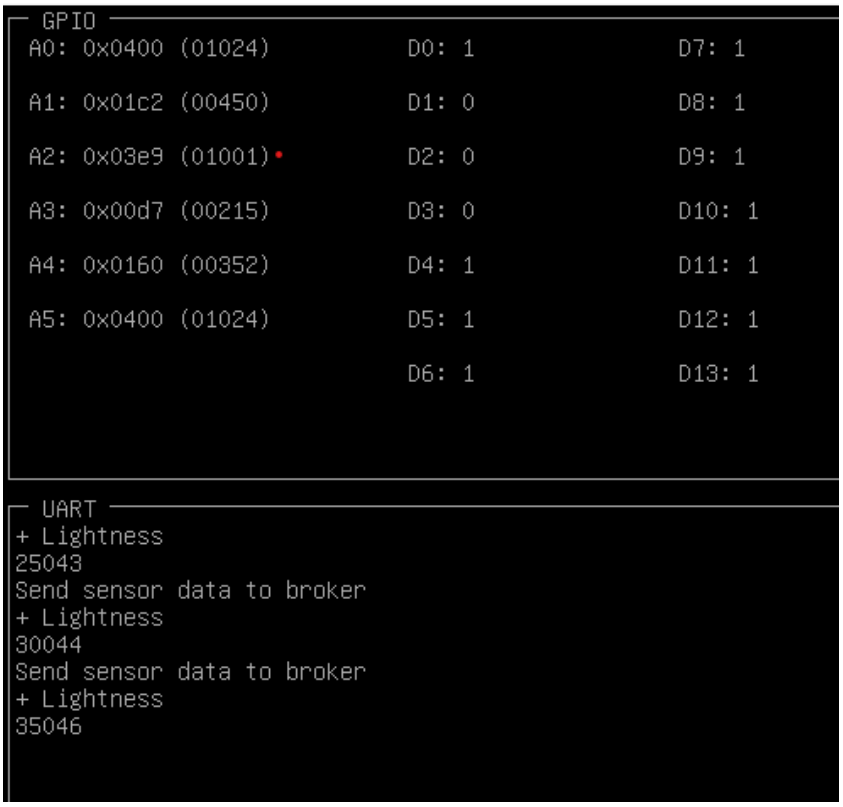
8.2. Podniesienie rolety

W celu podniesienia rolety sporządzono ponownie podobny prosty program który wysyłał polecenie *PUBLISH* na węzeł będący skrótem. Skrót ten był nasłuchiwany przez węzeł o `ID 502`, który umożliwiał sterowanie roletami. Fragment kodu wysyłającego znajduje się poniżej:

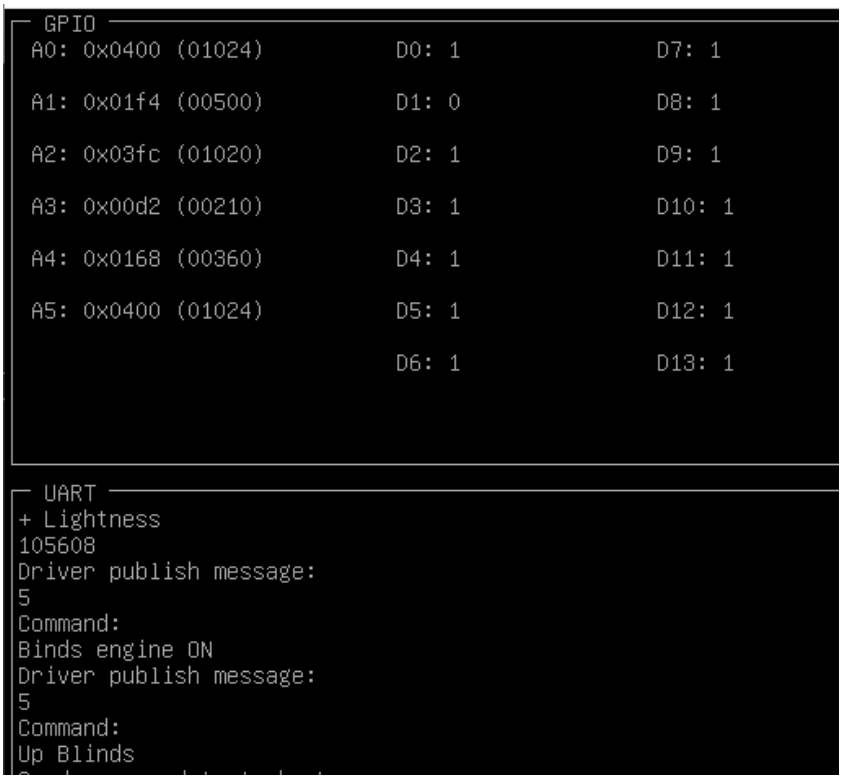
```
1     void test3(){
2     int len = 0;
3     // Test4
4
5     // Message4
6     char* path3 = "/mieszkanie/salon/okno1/roleta";
7     char* spath3 = encodePath(path3);
8     char* query3 = newQuery(spath3);
9
10    query3 = addWildcard(query3,COMMAND);
11
12    char* message3 = makePublishMessage(query3);
13
14    free(spath3);
15
16    len = packetLen(message3) + 2;
17    message3[len - 1] = TURN_ON_ENGINE_OF_BLINDS;
18
19    printf("length:␣%d␣\n",len);
20
21    sendMessage(message3,len);
22
23    free(message3);
24
25    // Message4
26    char* path4 = "/mieszkanie/salon/okno1/roleta";
27    char* spath4 = encodePath(path4);
28    char* query4 = newQuery(spath4);
29
30    query4 = addWildcard(query4,COMMAND);
31
32    char* message4 = makePublishMessage(query4);
33
34    free(spath4);
35
36    len = packetLen(message4) + 2;
37    message4[len - 1] = BLINDS_UP;
38
39    printf("length:␣%d␣\n",len);
40
41    sendMessage(message4,len);
42
43    free(message4);
44
45 }
```

Wysłano dwa rozkazy które są zdefiniowane w deklaracji programu (`BLINDS_UP` i `TURN_ON_ENGINE_OF_BLINDS`). Pierwszy z tych rozkazów służył do ustanowienia kierunku w którym żaluzje mają się poruszać (podnosić się lub opuszczać). Wybrano podnoszenie. Dodatkowo aby rozpocząć samo podnoszenie należało włączyć silnik zatem należało wysłać polecenie uruchamiające silnik.

W celu porównania zrobiono zrzut ekranu symulatora przed wykonaniem programu testowego:



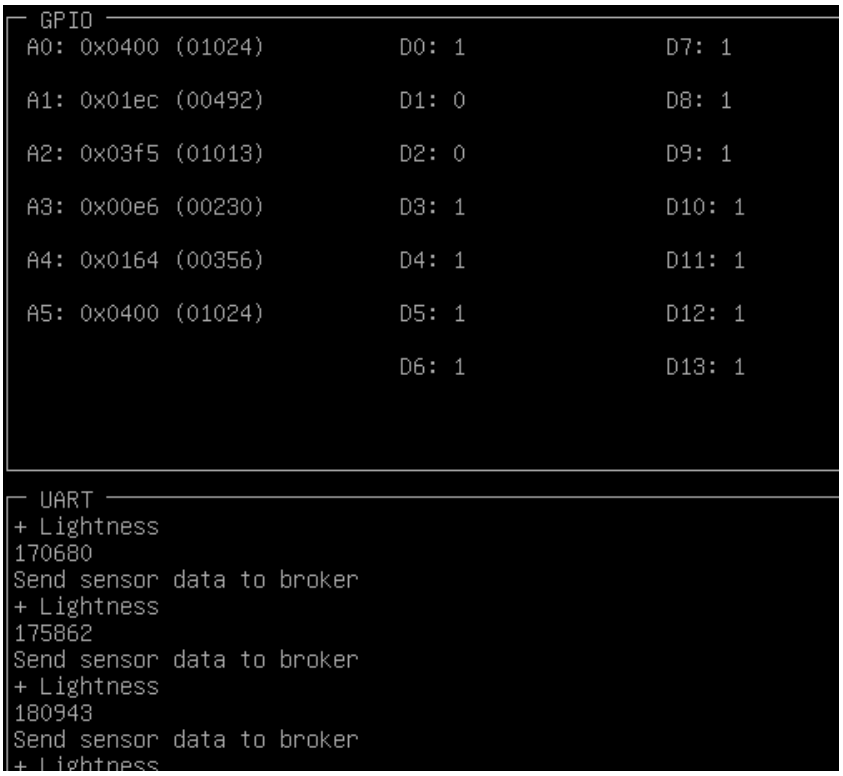
Następnie skompilowano i uruchomiono program. Efekty można zobaczyć poniżej:



Pin **D2** odpowiadający za działanie silnika oraz **D3** odpowiadający za kierunek działania silnika są w stanie wysokim. Oznacza to iż zgodnie z przyjętą konwencją roleta ta jest podnoszona. Osiągnięto efekt zgodny z

oczekiwaniami. Widać też na monitorze portu szeregowego, iż odebrano polecenia włączenia silnika (`Binds engine ON` oraz *Up Blinds*).

Następnie po ok 5 sekundach uzyskano taki efekt:



Zadziałało tutaj zabezpieczenie w węźle uniemożliwiające za długie działanie silnika. Uzyskany efekt jest zgody z oczekiwaniami i potwierdza możliwość sterowania węzłem za pomocą wiadomości *PUBLISH*.