Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Obiekty Internetu Rzeczy

(projekt)

Aplikacja węzła Internetu Rzeczy z interfejsem CoAP

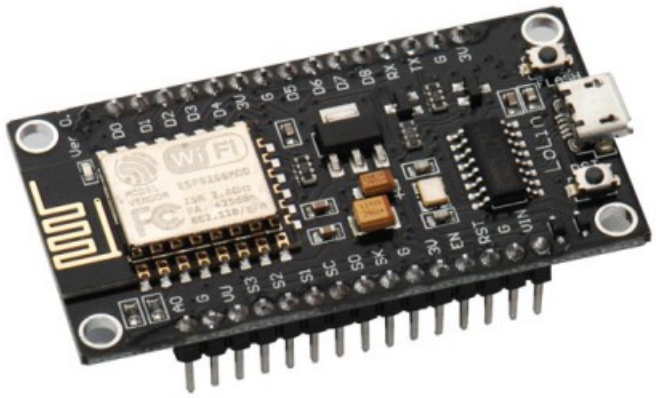
Pierczyk Krzysztof, Troć Patryk

Warszawa, 26 stycznia 2021

Spis treści

1 Wstęp

Celem projektu było stworzenie aplikacji serwerowej dla obiektu Internetu Rzeczy działającej w oparciu o autorską lub dostępną publicznie implementację protokołu CoAP (ang. *Constrained Application Protocol*). Platformą docelową został moduł **NodeMCU** w wersji trzeciej wyposarzony w układ SoC ESP8266 oraz 4MiB pamięci Flash dołączonej za pośrednictwem interfejsu Quad SPI. Niewątpliwą zaletą urządzenia jest zintegrowany moduł WiFi w standardzie 802.11b/g/n. Na płytce umieszczony został układ CH340 umożliwiający komunikację z wykorzystaniem protokołu USB $\leftrightarrow$ UART.



Rysunek 1: Płytka rozwojowa NodeMCU w wersjii trzeciej

Istnieją dwie zasadnicze możliwości programowania układów z rodziny ESP. Pierwsza z nich to dostosowany do możliwości platformy interfejs Arduino. Dostępny jest on do pobrania z poziomu Arduino IDE i umożliwia wykorzystanie bogatego zbioru bibliotek tworzonego przez społeczność zgromadzoną wokół platformy. Drugą z opcji jest posłużenie się dostarczanym przez producenta układu ESP8266 - firmę Espressif - zbiorem narzędzi dystrybuowanym pod nazwą ESP8266-RTOS-SDK **sdk**. Poza kompilatorem, skryptami linkera oraz implementacją biblioteki standardowej C SDK dostarcza także całą gamę sterowników, implementacji popularnych protokołów komunikacyjnych i szyfrujących a także narzędzia umożliwiające sprawne zarządzanie projektem oraz debugowanie. Ze względu na osobiste preferencje autorów w projekcie zdecydowano się wykorzystać platformę SDK. Prace na projektem podzielono na cztery etapy:

1. implementacja protokołu
2. stworzenie aplikacji serwerowej
3. opracowanie i przeprowadzenie testów
4. stworzenie dokumentacji

Ostatecznym efektem projektu jest oprogramowanie spełniające wszystkie postawione przed nim wymagania funkcjonalne.

2 Implementacja protokołu

Po przeanalizowaniu potencjalnych profitów z obydwu podejść do implementacji protokołu CoAP zdecydowano się na wariant pośredni. Punktem wyjściowym prac stała się popularna biblioteka `libcoap` **libcoap** stworzona przez Olafa Bergmanna. Poza realizacją bazowego standardu RFC7252 inkorporuje ona również standardy pochodne, m.in. RFC7641 (mechnizm obserwacji zasobów), RFC7959 (transfer blokowy), RFC8132 (metody PATCH i FETCH) oraz inne. Biblioteka została napisana w sposób multiplatformowy. Jest ona kompatybilna nie tylko z interfejsami programistycznymi systemów klasy desktop jak *Windows* czy *POSIX*, ale dostarzca także porty dla systemu *Contiki* oraz popularnego stosu TCP/IP *lwIP*. Argumentem przemawiającym za wyborem gotowej implementacji była możliwość zapoznania się z podejściem do wdrażania standardu przez osoby bardziej doświadczone oraz możliwość potencjalnego wykorzystania znajomości tej popularnej biblioteki w życiu zawodowym.

Oryginalny kod źródłowy postanowiono zmodyfikować tak, aby dopasować go do wybranej platformy sprzętowej. Oznaczało to usunięcie elementów realizujących multiplatformowość oraz próbę poprawienia fragmentów mających szczególny wpływ na zużycie zasobów. Zadanie to zostało ułatwione przez fakt częściowej zgodności interfejsu programistycznego dostarczanego przez ESP-RTOS-SDK ze standardem *POSIX*. Ponadto, ze względu na ograniczenia czasowe, zdecydowano usunąć z biblioteki mechanizmy szyfrujące. Najważniejszą decyzją projektową było postawienie na **pełną implementację standardów RFC7252, RFC7641 oraz RFC7959**. Chociaż wykracza to poza zakres wymagań projektowych uznano, że możliwość holistycznego zrozumienia protokołu przełoży się na poszerzenie świadomości ogólnych zasad rządzących standardami komunikacyjnymi.

Jako że przyjęte założenie wymagały ingerencji w każdy fragment oryginalnego kodu postanowiono także poprawić oryginalną dokumentację. Jak w wielu projektach otwartoźródłowych jest ona niejednolita a w wielu miejscach po prostu wybrakowana. Jej modyfikacja nie stanowiła tylko zaspokojenia perfekcjonistycznych potrzeb autorów, ale stała się także swego rodzaju weryfikatorem zrozumienia mechanizmu działania biblioteki.

2.1 Funkcjonalność

Jak zaznaczono na wstępie celem projektu była pełna implementacja standardów RFC7252 **rfc_coap**, RFC7641 **rfc_observer** i RFC7959 **rfc_block**. W trakcie prac postanowiono zrezygnować z niektórych elementów, które w opinii autorów sprowadzają się do rolii technicznych detali. Zaliczają się do nich niektóre kody opcji oraz odpowiedzi. Kluczowe fragmenty protokołu obejmujące m.in. format przesyłanych wiadomości, enkodowanie i dekodowanie pól opcji, transwer blokowy czy obserwację zasobów zostały w pełni zrealizowane. Ponadto biblioteka umożliwia wykorzystanie opisanego w RFC6690 **core_link** zasobu *well-known/core* do odkrywania dostępnych na serwerze zasobów.

2.2 Architektura - struktury danych

Biblioteka `libcoap` została napisana w całości w języku C. Jej główny koncept opiera się na manipulacji jawnie zdefiniowanymi strukturami danych reprezentującymi poszczególne obiekty związane z protokołem (jak np. zasób, obserwator, sesja). Elementem centralnym jest struktura `coap_context_t` zawierająca pełny zbiór informacji na temat stanu klienta/serwera CoAP. Wszystkie operacje natury stanowej odwołują się do instancji tej

struktury w celu ustalenia parametrów sesji, zajętości kolejki retransmitowanych wiadomości, czy historii generowanych kodów wiadomości (ang. *message ID*). Takie podejście sprawia, że sama biblioteka nie posiada stanu wewnętrznego, a co za tym idzie jej funkcje mają charakter reentrantny. Dzięki temu możliwe jest uruchomienie na jednej platformie kilku niezależnie działających wątków wykorzystujących protokół CoAP. Interakcja z biblioteką od strony programisty jest bardzo przejrzysta i sprowadza się kolejno do:

1. stworzenia instancji struktury opisującej kontekst
2. zadeklarowania portów, na których nasłuchiwał będzie serwer
3. zarejestrowania zasobów dostępnych na serwerze
4. zarejestrowania funkcji obsługujących zawartość zapytania o zasoby (ang. *resource handlers*)
5. cyklicznego wywoływania funkcji `coap_run_once()`

Jako że, jak powiedziano, cały zamysł implementacji kładzie szczególny nacisk na manipulację kilkoma kluczowymi strukturami danych, następne podrozdziały skupią się na ich opisaniu i określeniu ich miejsca w kompozycji protokołu.

2.2.1 Struktura coap_context_t

Poniższy listing przedstawia pola struktury `coap_context_t`. Ich omówienie powinno rzucić jaśniejsze światło na jej rolę w całym projekcie. Wskaźnik `app` przechowuje adres bloku danych użytkownika. Nie jest on wykorzystywany przez bibliotekę. Programista może zdecydować, by umieścić w nim dane, które będą mogły być współdzielone poprzez instancje serwera/klientów działajacych w obrębie pojedynczego kontekstu. Tablice `resources` oraz `unknown_resources` przechowują instancje struktury `coap_resource_t` (opisanej w dalszej części pracy) odnoszących się do zasobów umieszczonych na serwerze. Wyodrębnienie zbioru zasobów nienazwanych ma pomóć w radzeniu sobie z zapytaniami o zasoby nieznane serwerowi w czasie ich obsługi. Tablica `sendqueue` przechowuje pakiety, które oczekują na potwierdzenie (pakiet ACK). Każda z nich posiada własny stempel czasowy oraz liczbę dotychczasowych retransmisji. Wartości stempli są przechowywane relatywnie do wartości pola `sendqueue_basetime`.

Najważniejszymi elementami kontekstu są tablice `endpoint` oraz `sessions`. Ich elementy reprezentują kolejno gniazda, na których nasłuchuje serwer zarejestrowany w danym kontekście oraz otwarte sesje pomiędzy hostem a odległym serwerem. Strktura kontekstu protokołu została zaprojektowana tak, aby zmaksymalizować elastyczność jej użycia. W tym celu zawiera ona wskaźniki do funkcji odpowiedzialnych za obsługę przychodzących wiadomości określonego typu (NACK, RST) oraz za sam mechanizm sieciowy. Dzięki temu możliwa jest ich dynamiczna podmiana w trakcie działania systemu.

Ostatni segment pól stanowią parametry kontekstu. Dzięki dostosowaniu zmiennej `known_options` możemy ustalić jakie kody opcji będą rozpoznawane w ramach danego kontekstu, a jakie odrzucane. `session_timeout` oraz `max_idle_sessions` odpowiadają z kolei za politykę serwera względem utrzymywania otwartych sesji.

```
typedef struct coap_context_t {

    void *app;

    /* ---------------- Context's state ---------------- */

    struct coap_resource_t    *resources;
    struct coap_resource_t    *unknown_resource;
    coap_queue_t              *sendqueue;
    coap_tick_t                sendqueue_basetime;
    coap_endpoint_t           *endpoint;
    coap_session_t            *sessions;
    uint16_t                   message_id;

    /* ----------- Context-specific routines ----------- */

    coap_response_handler_t response_handler;
    coap_nack_handler_t     nack_handler;
    coap_ping_handler_t     ping_handler;

    ssize_t (*network_send)(
        coap_socket_t *sock,
        const coap_session_t *session,
        const uint8_t *data,
        size_t datalen);
    ssize_t (*network_read)(
        coap_socket_t *sock, struct coap_packet_t *packet);


    /* -------------- Context's parameters ------------- */

    coap_opt_filter_t known_options;
    unsigned int      session_timeout;
    unsigned int      max_idle_sessions;

} coap_context_t;
```

2.2.2 Struktura coap_resource_t

Jednym z najważniejszych pojęć przewijających się w kontekście protokołu coap jest **zasób**. W implementacji `libcoap` obiekt ten opisywany jest przez osobną strukturę - `coap_resource_t`. Podobnie jak w przypadku `coap_context_t` jej pierwszym polem jest wskaźnik do orbitralnego pola danych. Użytkownik może go wykorzystać, aby odwołać się do informacji powiązanych z zasobem z poziomu obsługi zapytań (informacją taką może być np. reprezentacja zasobu w pamięci).

Kolejnym elementem, który pojawia się w strukturze jest tablica funkcji. Zawiera ona wskaźniki do procedur obsługujących zapytania poszczególnych typów odnoszące się do danego zasobu (kolejno GET, POST, PUT, DELETE, FETCH, PATCH i IPATCH). Metody te można rejestrowac w ramach zasobu dzięki funkcji `coap_register_handler()`. Metody te musze posiadać określoną sygnaturę. Ich zadaniem jest odpowiednie skonfigurowanie wiadomości zwrotnej do klienta, przy czym część formatowania odbywa się automatycznie przed wywołaniem handlera. Należy pamiętać, że metody te sa wywoływane wewnątrz procedury `coap_run_once()` co oznacza, że ich wykonanie odbywa się bez

potrzeba tworzenia nowego wątku. Pole `hh` stanowi zmienną pomocniczą wykorzystywaną przez mechanizm haszujący w przypadku zagnieżdżania zasobów w tablicy. W tym miejscu warto wspomnieć, że `libcoap` wykorzystuje gotową implementację tablic haszujących autorstwa Troya D. Hansona.

```
typedef struct coap_resource_t {

    void *user_data;

    /* ------------------- Handlers --------------------- */

    coap_method_handler_t handler[7];

    /* --------------------- Helper ---------------------- */

    UT_hash_handle    hh;

    /* --------------------- Flags ----------------------- */

    unsigned int dirty:1;
    unsigned int partiallydirty:1;
    unsigned int observable:1;
    unsigned int cacheable:1;
    unsigned int is_unknown:1;
    int          flags;

    /* --------------- Resource--------------------------- */

    coap_attr_t      *link_attr;
    coap_str_const_t *uri_path;

    /* ------------ Observers-related info ------------- */

    coap_subscription_t *subscribers;
    unsigned int         observe;

} coap_resource_t;
```

Zestaw flag bitowych zawarty w strukturze wykorzystywany jest przede wszystkim w przypadku obserwacji zasobów przez klientów. Łańcuch URI identyfikujący zasób jest również częścią struktury `coap_resource_t`. Ponadto, zgodnie ze standardem każdy zasób może posiadać arbitralny ciąg opisujących go atrybutów umieszczanych w tablicy `link_attr`. Ostatnim elementem związanym z zasobami są obserwatorzy. Aby zgodnie z **rfc_observer** serwer mógł wysyłać wiadomości z rosnącymi wartościami pola *Observe*, ostatnia użyta wartość jest przechowywana w zmiennej `observe`.

2.2.3 Struktura coap_endpoint_t

Obiekty typu `coap_endpoint_t` reprezentują gniazda sieciowe, na których nasłuchuje serwer. W ramach jednego kontekstu może być ich zarejestrowana dowolna ilość. Pole `next` służy do tworzenia list jednokierunkowych. `default_mtu` określa wielkość MTU (ang. *Maximum Transimission Unit*) w bajtach dla danego interfejsu.

```
typedef struct coap_endpoint_t {

    struct coap_endpoint_t *next;
    struct coap_context_t *context;

    uint16_t        default_mtu;
    coap_socket_t   sock;

    coap_address_t  bind_addr;
    coap_session_t *sessions;

} coap_endpoint_t;
```

`sock` oraz `bind_addr` stanowią element łączący bibliotekę z infrastrukturą sieciową systemu. Ostatecznie w strukturze obecna jest również lista aktywnych sesji.

2.2.4 Struktura coap_session_t

```
typedef struct coap_session_t {

    struct coap_session_t *next;
    struct coap_context_t *context;
    void                  *app;

    /* --------------- Basic session info -------------- */

    coap_session_type_t  type;
    coap_session_state_t state;
    unsigned             ref;

    /* -------------- Session's parameters ------------- */

    unsigned           mtu;
    unsigned int       max_retransmit;
    coap_fixed_point_t ack_timeout;
    coap_fixed_point_t ack_random_factor;

    /* ---------------- Endpoints' info ---------------- */

    coap_address_t          remote_addr;
    coap_address_t          local_addr;
    coap_socket_t           sock;
    struct coap_endpoint_t *endpoint;

    /* ----------------- Messages' info ---------------- */

    uint16_t             tx_mid;
    uint8_t              con_active;
    struct coap_queue_t *delayqueue;
    coap_tick_t          last_rx_tx;
    coap_tick_t          last_tx_rst;

} coap_session_t;
```

Tak jak struktura `coap_context_t` jest centralnym punktem implementacji protokołu jako całości, tak struktura `coap_session_t` (wraz z opisaną poniżej `coap_pdu_t`) jest środkiem ciężkości mechanizmów komunikacji. Rola pierwszych trzech pól struktury może zostać wywnioskowana ze wcześniejszych opisów. Pole `type` determinuje charakter obiektu. Może on okreslać, czy sesja została stworzona przez lokalnego hosta w ramach zapytania klienckiego, czy na skutek przyjęcia zapytania do serwera. `state` dopełnia tę informację określając, czy sesja związana jest aktualnie z jakimś połączeniem internetowym czy też nie. Zmienna `ref` stanowi licznik odwołań do obiektu sesji z globalnej kolejki wiadomości oczekujących na potwierdzenie. Gdy wartość licznika spadnie do zera sesja jest uznawana za istniejąca w trybie IDLE. Nie jest jest usuwana z systemu od razu, gdyż może być wykorzystana przy ponownym zapytania/odpowiedzi. Jeśli jednak pozostanie ona w stanie IDLE zbyt długo, zostanie usunięta na skutek wywołania `coap_run_once()`.

Nazwy parametrów z sekcji *Session's parameters* wydają się być autodeskryptywne. Jedynym wartym wspomnienia jest `ack_random_factor`. Jest to współczynnik słóżący do generowania pseudolosowych okresów oczekiwania pomiędzy kolejnymi retransmisjami, który jest wymagany przez standard. Sekcja *Endpoints' info* zawiera zmienne wiążące sesję z systemowym interfejsem sieciowym podobnie jak miało to miejsce w przypadku `coap_endpoint_t`.

Ostatni sekcja zawiera informacje na temat wysyłanych pakietów. `tx_mid` jest to MID ostatniego wysyłanego pakietu. `con_active` określa ilość wiadomości typu CON oczekujących na potwierdzenie (ACK). Parametry `last_rx_tx` oraz `last_tx_rst` stanowią stemple czasowe wysyłanych za pośrednictwem sesji wiadomości. Najciekawsza jest jednak tablica `delayqueue`. Gdy sesja nada wiadomość typu CON odsyła ją do globalnej kolejki znajdującej się w instancji kontekstu. Kolejka ta ma jednak ograniczoną pojemność (ograniczoną liczbę równolegle utrzymywanych, niezatwierdzonych wiadomości). Jeżeli sesja nie może umieścić w kolejce kolejnej wiadomości typu CON, wstrzymuje się ona z jej nadaniem. Opóźnione w ten sposób pakiety są oddelegowywane do kolejki `delayqueue` i rozpatrywane przy następnym wywołaniu `coap_run_once`.

2.2.5 Struktura coap_subscription_t

```
typedef struct coap_subscription_t {

    struct coap_subscription_t *next;
    coap_session_t *session;

    /* ------------- Basic subscriber info ------------- */

    unsigned int   non_cnt:4;
    unsigned int   fail_cnt:2;
    unsigned int   dirty:1;
    unsigned int   has_block2:1;
    coap_block_t   block2;
    coap_string_t *query;

    /* ------------------ Token info ------------------- */

    size_t token_length;
    unsigned char token[8];

} coap_subscription_t;
```

Opis klienta obserwującego dany zasób również został zdefiniowany w jawnie określnej strukturze. Zawiera ona przede wszystkim kluczowe informacje pozwalające konstruować pakiety wysyłane po zaistnieniu modyfikacji zasobu. Flaga `dirty` ustawiana jest, gdy notyfikacja z jakiegoś powodu nie mogła zostać wysłana. `fail_cnt` stanowi z kolei licznik retransmisji tych notyfikacji. Maksymalna ilość notyfikacji, które mogą zostać wysłane do klienta bez potwierdzenia (ACK) przechowywany jest w liczniku `non_cnt`.

Jeżeli zapytanie o wpisanie klienta do listy obserwatorów zostało wysłane z ustawioną opcją *Block2* zostanie to odwzorowane z użyciem flagi `has_block2`. W strukturze przechowywany jest także obiekt `coap_block_t`, który śledzi rozmiar wysyłanych bloków oraz indeks następnego bloku do nadania w ramach notyfikacji. Informacje dotyczące obserwatora zamykają token oraz zapytanie (ang. *querry*) użyte w pakiecie subskrybującym.

2.2.6 Struktura coap_pdu_t

```
typedef struct coap_pdu_t {

    /*
    *                  PDU's memory layout
    *
    * ------------------------------------------------------
    * |<-header->|<-token->|<-options->|0xFF|<-payload->|
    * ------------------------------------------------------
    */

   /* ------------------ Header info ------------------- */

    uint8_t  type;
    uint8_t  code;
    uint16_t tid;
    uint16_t max_delta;
    uint8_t  token_length;

    /* ------------------- Size info ------------------- */

    size_t alloc_size;
    size_t used_size;
    size_t max_size;

    /* ---------------------- Data --------------------- */

    uint8_t *token;
    uint8_t *data;

} coap_pdu_t;
```

`coap_pdu_t` stanowi opis pakietu, który zostanie stworzony. Przed wysłaniem wiadomości w ramach aktywnej sesji jego zawartość zostanie przeanalizowana pod kątem zajętości pamięci i, jeżeli nie przekracza ona wartości dopuszczalnego MTU, przetransformowana do postaci bufora binarnego. Pierwsze dwa pola - `type` i `code` - odpowiadają wartościom z nagłówka pakietu CoAP. `tid` reprezentuje MID, natomiast `max_delta` najwyższy indeks opcji wpisany do pakietu. Najważniejszymi polami są `token` oraz `data`. Pierwszy z nich wskazuje bufor pamięci, w którym umieszczony jest token. Jak pokazano na li-

stingu wskaźnik ten odnosi się do pierwszego bajtu za nagłówkiem. W obszarze pamięci pomiędzy `token` a `data` znajdują się zapisane opcje. Na tym etapie są one już zakodowane do postaci binarnej z wykorzystaniem kodowania różnicowego (ang. *delta encoding*). Obszar opcji jest zamykany poprzez znacznik $0xFF$ w momencie wpisania pierwszego bajtu danych do pakietu. Pola `alloc_size`, `used_size` oraz `max_size` określają kolejno ilość pamięci zaalokowaną na rzecz pakietu, ilość wykorzystanej pamięci (z zaalokowanej publi) oraz maksymalny rozmiar pakietu (bez uwzględnienia nagłówka).

2.2.7 Struktury danych związana z opcjami

Ostatnimi z kluczowych struktur danych wykorzystanych w projekcie są te powiązane z enkodowaniem i dekodowaniem pól opcji. Mamy w tym przypadku do czynienia z trzema takimi strukturami. Pierwsza z nich - `coap_optlist_t` - reprezentuje wysokopoziomowy opis listy opcji (które będą wpisane lub zostały odczytane z pakietu).

```
typedef struct coap_optlist_t {

  struct coap_optlist_t *next;

  uint16_t number;
  size_t length;
  uint8_t *data;

} coap_optlist_t;
```

Nazwy zawartych w niej zmiennych wydają się być wystarczająco wymowne. Warto jednak zauważyć, że pole `number` odnosi się do bezwzględnego identyfikatora opcji. Gotową listę opcji można przekazać do funkcji `coap_add_optlist_pdu()`, która przekonwertuje opis opcji do postaci binarnej. Do wstawiania kolejnych opcji do listy służy funkcja `coap_insert_optlist()`. Po każdy wstawieniu opcji lista jest sortowana zgodnie z rosnącymi numerami opcji.

```
typedef struct {

    coap_opt_t *next_option;

    size_t length;
    uint16_t type;

    unsigned int bad:1;
    unsigned int filtered:1;
    coap_opt_filter_t filter;

} coap_opt_iterator_t;
```

Następne dwie struktury danych służą do dekodowania opcji z pakietów w postaci binarnej. `coap_opt_iterator_t` służy do iterowania po zakodowanych opcjach. Opcje takie są opisywane przez strukturę `coap_option_t`. W czasie dekodowania iterator parsuje kolejne segmenty danych (utrzymując wskaźnik do następnego segmentu w zmiennej `next_option`) do opisu w postaci tej struktury.

```
typedef struct {

    uint16_t delta;
    size_t length;
    const uint8_t *value;

} coap_option_t;
```

2.2.8 Struktury danych - podsumowanie

Opisane struktury są kluczowymi do zrozumienia mechanizmu działania wykorzystanej implementacji protokołu CoAP. Nie są one jednak jedyne. Decyzja o dostarczeniu pełnej funkcjonalności protokołu poskutkowała kodem, który wraz z dokumentacją liczy ponad 15′000 linii. W naturalny sposób przekłada się to na dziesiątki pomniejszych struktur oraz funkcji pomocniczych, których nie sposób ująć w sprawozdaniu o sensownych ramach.

Instancje obiektów są powiązane w trakcie działania programu poprzez wywołania funkcji z biblioteki `libcoap`. Zarówno twórcy biblioteki jak i my (poprzez wprowadzone modyfikacje) dołożyliśmy starań aby zmaksymalizować wygodę korzystania z dostarczanych rozwiązań. Bezpośrednia ingerencja w zdefiniowane struktury danych przez użytkownika nie powinna mieć miejsca. Dla wszystkich przewidzianych mechanizmów dostarczone zostały odpowiednie funkcje.

2.3 Architektura - przepływ danych

Przepływ sterowania z wykorzystaniem `libcoap` został zaprojektowany tak, aby w jak największym stopniu odciążyć programistę w aspektach zależnych od protokołu. Projekt prostego serwera zamyka się w kilkudziesięciu liniach kodu. Niniejszy podrozdział przedstawia podstawowe API, z którym styka się programista w przypadku rutynowych zadań.

2.3.1 Inicjalizacja stosu danych

Pierwszym krokiem na drodze do wykorzystania biblioteki jest inicjalizacja kontekstu. W przypadku aplikacji typu serwer należy również zadeklarować interfejsy sieciowe, na których program będzie nasłuchiwał.

```
// Initialize CoAP's contex structure
coap_context_t *ctx = coap_new_context(NULL);
if (!ctx)
    exit(1);

// Prepare interface for listening
coap_address_init(&serv_addr);
serv_addr.addr.sin.sin_addr.s_addr = INADDR_ANY;
serv_addr.addr.sin.sin_family      = AF_INET;
serv_addr.addr.sin.sin_port        = htons(PORT);

// Create UDP endpoint
coap_endpoint_t *ep = coap_new_endpoint(ctx, &serv_addr);
if (!ep)
    exit(1);
```

2.3.2 Zasoby

Po zainicjalizowaniu kontekstu możliwe jest zarejestrowanie zasobów dostępnych na serwerze. Niniejszy listing pokazuje przykładowy przebieg rejestracji zasobu `time`. W czasie tworzenia instancji zasobu możliwe jest ustawienie atrubutów oraz możliwości obserwowania.

```
// Create a new resource
coap_resource_t *resource = coap_resource_init(coap_make_str_const("
    time"), 0);
if(!resource){
    coap_delete_all_resources(context);
    exit(1);
}

// Document a resource with attributes (describe resource when GET /.
    well-known/core is called)
coap_add_attr(resource,    coap_make_str_const("ct"),
    coap_make_str_const("\"plain text\""), 0);
coap_add_attr(resource,    coap_make_str_const("rt"),
    coap_make_str_const("\"time\""), 0);
coap_add_attr(resource,    coap_make_str_const("if"),
    coap_make_str_const("\"GET\""), 0);

// Register resource's data
uint32_t *time = malloc(sizeof(uint32_t));
coap_resource_set_userdata(resource, time);

// Register handlers for methods called on the resourse
coap_register_handler(resource, COAP_REQUEST_GET, hnd_get);

// Set the resource as observable
coap_resource_set_observable(resource, 1);

// Add the resource to the context
coap_add_resource(context, resource);
```

Procedura obsługująca zapytania związane z danym zasobem (tu: `hnd_get`) powinna mieć pokazaną niżej sygnaturę.

```
void hnd_get(
    coap_resource_t *resource,
    coap_session_t *session,
    coap_pdu_t *request,
    coap_binary_t *token,
    coap_string_t *query,
    coap_pdu_t *response
);
```

Efektem działania funkcji (z punktu widzenia biblioteki) powinno być ustawienie porządanego kodu odpowiedzi, opcji, oraz danych. To jak odpowiedzieć zdecyduje się serwer zależy w pełni od projektanta aplikacji. W przypadku zapytania GET może on ustawić kod błędu, lub zwrócić rządany zasób. Może też (jak w przypadku zasobów o długim czasie dostępu) zdecydowac się na odesłanie pustej odpowiedzi i uruchomienie wewnętrz-

nych mechanizmów serwera, które w sposób asynchroniczny przygotują i wyślą pakiet z reprezentacją zasobu. Implementacja biblioteki nie stawia w tym kontekście żadnych wymagań.

Przedstawione wyżej funkcje są najczęściej używanymi w kontekście zasobów. Jedyną nieukazaną tam jest `coap_resource_notify_observers()`. Funkcja ta powinna zostać wywołana zawsze, gdy serwer zmieni stan zasobu. Ustawi ona odpowiednią flagę, dzięki której następna iteracja `coap_run_once()` roześle powiadomienia do zadeklarowanych obserwatorów.

2.3.3 Tworzenie pakietów

Chociaż w niektórych przypadkach serwer jest w stanie operowac jedynie z wykorzystaniem odpowiedzi przygotowanych przez funkcje biblioteczne przed wywołaniem skoku do dedykowanego handlera, to jednak w niektórych przypadkach (jak zapytania o zasób o długim czasie dostępu) odpowiedź musi być wysyłana asynchronicznie. Ponadto manualnego konstruowania pakietów nie da się uniknąć w przypadku zapytań klienckich. Poniższy listing ukazuje procedurę tworzenia właśnie tego typu pakietu. Sesję kliencką tworzy się poprzez procedurę `coap_new_client_session()`. Pierwszym argumentem wywołania jest w tym przypadku zainicjalizowany wcześniej kontekst, drugim interfejs lokalny przez który wysłane zostanie zapytanie (w przypadku NULL'a zastosowany zostanie IF_ANY), a trzecim adres serwera docelowego. Po pomyślnym stworzeniu sesji można przejść do budowy pakietu.

```
// Initialize server's address
coap_address_init(&server);
server.addr.sa.sa_family = AF_INET;
server.addr.sin.sin_addr = server_ip_address;
server.addr.sin.sin_port = htons(5683);

// Initialize client session
coap_session_t *session = coap_new_client_session(context, NULL, &
    server);
if (!session)
    exit(1);

// Initialize PDU
coap_pdu_t *pdu = coap_pdu_init(
    message_type,
    request_code,
    coap_new_message_id(session),
    coap_session_max_pdu_size(session)
);
if (!pdu)
    return 0;

// Add token to the PDU
if (!coap_add_token(pdu, sizeof(token), (unsigned char*)&token)) {
    exit(1);
```

Budowę pakietu należy rozpocząć od wywołania funkcji `coap_pdu_init()`. Podać należy typ wiadomości oraz kod zapytania. MID może zostać wygenerowane automatycznie na bazie wykorzystywanej sesji. Po inicjalizacji PDU należy zawsze w pierwszej kolejności

dodać token.

Kolejny listing przedstawia dodawanie do PDU zestawu opcji (tu: URI_PATH oraz URI_QUERY). Każda opcja musi zostać dodana do listy `optlist_chain`, która zawiera jej wysokopoziomowy opis. Dodanie do listy realizuje funkcja `coap_insert_optlist()`. Z kolei `coap_new_optlist()` tworzy instancję wysokopoziomowego opisu opcji.

```
char buf[1024];
char *sbuf = buf;
size_t buflen;
coap_optlist_t *optlist_chain = NULL;

// Add in the URI options
buflen = sizeof(buf);
int res = coap_split_path((const uint8_t*) uri,
    strlen(uri), sbuf, &buflen);
while (res--) {
    if (!coap_insert_optlist(
        &optlist_chain,
        coap_new_optlist(
            COAP_OPTION_URI_PATH,
            coap_opt_length(sbuf),
            coap_opt_value(sbuf)
        )))
        exit(1);
    sbuf += coap_opt_size(sbuf);
}



// Add in the QUERY options
buflen = sizeof(buf);
res = coap_split_query((const uint8_t*) query,
    strlen(query), sbuf, &buflen);
while (res--) {
    if (!coap_insert_optlist(
        &optlist_chain,
        coap_new_optlist(
            COAP_OPTION_URI_QUERY,
            coap_opt_length(sbuf),
            coap_opt_value(sbuf)
        )))
        exit(1);
    sbuf += coap_opt_size(sbuf);
}

// Add in options to the pdu
if (!coap_add_optlist_pdu(pdu, &optlist_chain))
    exit(1);
```

Po skonfigurowaniu łańcucha możliwe jest przekonwertowanie opcji do postaci binarnej poprzez wywołanie `coap_add_optlist_pdu()`. Funkcja ta sortuje odpowiednio listę opcji oraz zamienia indeksy absolutne na różnicowe (*delta coding*). Ręczne tworzenie pakietów daje się do pewnego stopnia zautomatyzować w przypadku odpowiedzi na zapytania. Biblioteka `libcoap` udostepnia pomocnicza funkcję `coap_add_data_blocked_response()`. Jest ona wywoływana najczęściej z wnętrza handlera GET. Analizuje ona zawartość zapy-

tania i ustawia odpowiednie opcje w pakiecie zwrotnym (np. MAX_AGE, OBSERVE, ...). Jeżeli zapytanie zawierało opcję BLOCK_2, to funkcja ta wstawi do pakietu odpowiedni fragment przekazanych jej danych. Sygnatura funkcji prezentuje się nastepująco:

```
void coap_add_data_blocked_response(
    struct coap_resource_t *resource,
    struct coap_session_t *session,
    coap_pdu_t *request,
    coap_pdu_t *response,
    const coap_binary_t *token,
    uint16_t media_type,
    int maxage,
    size_t length,
    const uint8_t* data
);
```

2.4 Architektura - podsumowanie

Przedstawiona część dostępnego API jest tylko niewielkim wycinkiem całości. Opisanie wszystkich zawartych mechanizmów wymagałoby znacznie szerszej dokumentacji. W naszej opinii prezentacja ta wystarczy jednak aby zapoznać się z koncepcją stojącą za implementacją `libcoap` a także aby zdobyć wiedzę potrzebną do napisania prostych aplikacji typu klient-serwer. Wyszczególnione zostały tu kluczowe zagadnienia dotyczące zarządzania sesją, tworzenia zasobów oraz konstrukcji pakietów. Pozostała część biblioteki stanowi solidne uzupełnienie tych mechanizmów o procedury i struktury danych, które upraszczają niektóre rutynowe zabiegi.

3 Implementacja serwera

Omówione wyżej rozwiązania programistyczne oraz funkcjnalności zapewnione dzięki bibliotekom pozwalają na stworzenie aplikacji obsługującej węzły czy serwer w sieci IoT działającej na protokole CoAP. W tym rozdziale chcielibyśmy omówić działanie zaimplementowanego przez nas serwera CoAP, czyli przedstawić jego ogólne działanie, a także opisac zasoby, które ten serwer udostępnia, i ich funkcjonalności.

3.1 Program główny

Poniżej omówię pliki z kodem źródłowym odpowiedzialne za implementację serwera CoAP. Wymienię główne funkcje i realizowane rozwiązania programistyczne.

3.1.1 coap_server_main.c

Program główny (main) zawiera się w pliku `coap_server_main.c`. Główna funkcja jest odpowiedzialna za uruchomienie tramsmisji *Wi-Fi* i połączenie z lokalną siecią, następnie przechodzimy do funkcji odpowiedzialnej za obsługę całego interfejsu CoAP. Później uruchamiany jest serwer UDP, po którego zakończniu następuje zatrzymanie działania serwera CoAP, a następnie rozłączenie się z siecią *Wi-Fi*. Całość wygląda następująco:

```
void app_main(){

    // Initialize NVS flash for other components' use
    ESP_ERROR_CHECK(nvs_flash_init());

    // LOG start of the Programm
    ESP_LOGI(TAG, "Connecting to WiFi AP...");

    // Connect via WiFi to the AP
    wifi_connect(EXAMPLE_ESP_WIFI_SSID, EXAMPLE_ESP_WIFI_PASS);

    // Create UDP server tasl
    xTaskCreate(coap_example_thread, "coap", 1024 * 10, NULL, 5, NULL
        );
    // Wait for udpp server task to finish
    main_handler = xTaskGetCurrentTaskHandle();
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

    // Disconnect from the AP
    wifi_disconnect();
}
```

Oprócz głównej funkcji mamy również zadeklarowane zmienne globalne potrzebne do połączenia modułu z siecią Wi-Fi (czyli nazwa sieci i hasło, które łatwo można zmienić przed kompilacją programu do naszych zapotrzebowań) oraz funkcja odpowiedzialna za obsługę protokołu CoAP, której inicjację mamy w pliku, który poniżej mamy zamiar omówić.

3.1.2 coap_server.c

W tym pliku mamy zawartą zainicjowaną funkcję *void coap_example_threadvoid * pvParameters*, która jest odpowiedzialna za serwer CoAP, odbieranie i przesyłanie danych do klienta, oraz kontrolę zasobów. Na początku jest inicjalizacja kontekstu i stosu danych (która została wcześniej przez nas omówiona), potem mamy opisaną pętlę główną serwera, która wygląda następująco:

```
// Run main processing loop
ESP_LOGI(TAG, "Beginning dispatch loop");
unsigned wait_ms = COAP_RESOURCE_CHECK_TIME * 1000;
while (1) {
        if (packet_loss_flag)
        {
                coap_debug_set_packet_loss("0%");
                packet_loss_flag=0;
        }
        // Server incoming and outcoming packages
        int result = coap_run_once(ctx, wait_ms);

        // Back to CoAP server initialization, when error occurs
        if (result < 0){
                coap_free_context(ctx);
                break;
        }
        // Decrement timeout if the last one was shorter than
           expected
        else if (result < wait_ms)
                wait_ms -= result;
        // Reset the timeout otherwise
        else
                wait_ms = COAP_RESOURCE_CHECK_TIME * 1000;
```

Na samaym początku jest wyłączane tracenie pakietów, które włączamy dopiero przy realizacji żądania GET metryki wysyłającej odpowiedź CON, lecz to omówimy później. Serwer w tej pętli uruchamia funkcję, która przetwarza wszystkie oczekujące pakiety do wysłania dla określonego kontekstu i czeka na przetworzenie wszystkich pakietów wejściowych przed powrotem. Jeśli pojawią się na tym etapie błędy, serwer jest cofany do etapu inicjalizacji. Następnie mamy kilka linii kodu, która kalibruje czas końcowy do rzeczywistego czasu, który jest realizowany w praktyce. Pętla ta jest tak na prawdę sercem naszego serwera i ona nadaje rytm działania.

3.1.3 coap_handlers

Ten plik kodu odpowiada za inicjację, charakterystykę zasobów i reakcje na wiadomości klienta ich dotyczące. Składa się z kilku funkcji:

1. `int resources_init(coap_context_t *context)` jest funkcją inicjującą zasoby w danym kontekście. Jest ona złożona z funkcji inicjujących poszczególne zasoby w naszym serwerze według wzoru, którego przykład przedstawiliśmy w podrozdziale `2.3.2 Zasoby`. Charakterystykę zasobów omówimy w osobnym podrozdziale

2. `void resources_deinit(coap_context_t *context)` odpowiedzialna za usunięcie zasobów z kontekstu.

3. `void hnd_get` odpowiedzialna za obsługę żądań GET ze strony klienta. Jej główne zadania i dane wejściowe również zostały omówione wyżej. Jest ona podzielona na kilka wariantów, w zależności od tego, jakiego zasobu dane żądanie dotyczy, kończy się ta funkcja wysłaniem przygotowanej przez serwer odpowiedzi. Jako przykład przedstawiam realizację dla jednego z zasobów:

```
// Handle ' GET /metrics/PUT_inputs' request
if( resource == coap_get_resource_from_uri_path(session->context,
    coap_make_str_const("metrics/GET_inputs")) ){

        char bufor[20];
        uint8_t size;
        size=snprintf(bufor, 14, "GET inputs: %d", GET_counter);
        // Send data with dedicated function
        coap_add_data_blocked_response(
                resource,
                session,
                request,
                response,
                token,
                COAP_MEDIATYPE_TEXT_PLAIN,
                0,
                size,
                (uint8_t *) bufor
        );
}
```

4. `void hnd_get` odpowiedzialna za obsługę żądań PUT ze strony klienta. Na podstawie otrzymanego payloadu wykonuje dane polecenia i wysyła adekwatną wiadomość do klienta: 204 (Changed) lub 400 (Bad Request). Zawiera też na końcu wspomnianą wcześniej funkcję `coap_resource_notify_observes()` odpowiadającą za rozesłanie powiadomienia do potencjalnych obserwatorów zasobów.

Poza wyżej wymienionymi funkcjami ten plik ma też zadeklarowane zmienne globalne wspomagające działanie zasobów.

3.1.4 rpn_stack

W tym pliku mamy zawarty algorytm obliczania wartości wyrażenia w notacji polskiej odwrotnej. Notacja ta jest zbudowana na śtosie"(który w praktyce jest tabelą liczb) i metodach odpowiedzialnych za zdejmowanie i kładzenie na stosie danych wartości. Główną funkcją jest poniższa funkcja:

```
//Calculate RPN for string expression and n variable
uint8_t getRPN(char * expression, uint8_t n)
{
    char * ch;          //pointer to parsing components
    char exp[30];       //string, when input expression is copied
    strcpy(exp, expression);
    makeEmpty();
    //parse first component from expression
    ch = strtok (exp," ");
```

```
while (ch!=NULL)
{
            //if component is a number
    if (ch[0]>=48 && ch[0]<=57)
       push(atoi(ch));
    //if component is a n variable
    else if (ch[0]=='n')
        push(n);
    //if component is a sign
    else
    {
        switch(ch[0])
                    {
                            case '+':
                                push(pop() + pop());
                                break;
                            case '-':
                                push(pop()-pop());
                                break;
                            case '*':
                                push(pop()*pop());
                                break;
                            case '/':
                                push(pop()/pop());
                                break;
                    }
    }
       //parse new sign
    ch = strtok (NULL, " ");
          }
          return pop();
    }
```

Funkcja ta wyciąga kolejne składniki z ciągu znaków przesłanego na wejściu i w zależności od tego, czy ma do czynienia ze zmienną n, czy ze znakiem działania, czy z liczbą wykonuje odpowiednie zadania na stosie: w przypadku liczby kładzie ją na stosie, w przypadku zmiennej kładzie jej przesłaną wartość na stosie, a w przypadku działania zdejmuje ze stosu 2 liczby, wykonuje dane działanie i kładzie wynik na stosie. Funkzja ta zakłada, że zapis w notacji został wykonany poprawnie, a także że argumenty i wynik działania jest liczbą całkowitą.

3.1.5 Program główny - podsumowanie

Wymienione zostały w tym podrozdziale funkcje, które są realizowane przez program główny naszego serwera oraz najważniejsze jego elementy. Dokładna semantyka stojąca za naszymi rozwiązaniami zostanie omówiona w następnym podrozdziale dotyczącym zasobów

3.2 Zasoby w serwerze

Zgodnie z treścią zadania przygotowaliśmy 5 zasobów realizujących konkretne zadania. Wśród nich jest 1 zasób odpowiedzialny za zbiór wyrażeń algebraicznych w notacji polskiej odwrotnej (zapis i obliczenia), 3 natomiast są metrykami (statystykami) dotyczą-

cymi przesyłanych datagramów pomiędzy klientem a serwerem. Wszystkie te zasoby są udostępniane przez piąty zasób o ścieżce *./well-known/core*.

3.2.1 RPN

Jest to zasób odpowiadający za obsługę zbioru wyrażeń algebraicznych w notacji polskiej odwrotnej. Jego ścieżka jest */rpn*. Współpracuje on z zadeklarowaną globalnie tablicą dwuwymiarową znaków i licznikiem zapisanych wyrażeń o następujących deklaracjach i inicjacjach:

```
#define RPN_MAX_SIZE 10
#define EXP_MAX_SIZE 30
char rpn_col[RPN_MAX_SIZE][EXP_MAX_SIZE]={NULL};
uint8_t rpn_expression_count = 0;
```

Zasób ten realizuje 3 rodzaje żądań:

1. GET - pobiera wszystkie wyrażenia ze zbioru i wysyła je do klienta. W jej przypadku należy w Uri-query wpisać *all*, aby żądanie zostało zrealizowane.

2. GET - wysyła dla podanej wartości n i numeru wyrażenia (wyr) wynik. Aby wywołać realizację tego żądania, należy zamieścić w Uri-query tekst: $wyr{=}x\&n{=}y$ lub $n{=}x\&wyr{=}y$, gdzie x i y są naszymi zmiennymi. Podajemy jedynie liczby całkowite, jeśli chodzi o wartość wyr numeracja wyrażeń w tablicy zaczyna się od 1. Jeśli zabraknie któregoś elementu, lub będzie nieprawidłowo podany, klient dostanie wiadomość o braku danego wyrażenia. Jeśli wartość wyr będzie większa od liczby zapisanych w zbiorze wyrażeń, również otrzymamy odpowiednią wiadomość.

3. PUT - przesyła daną wejściową do tablicy wyrażeń algebraicznych, jeśli nie jest przekroczony rozmiar tablicy. W przeciwnym przypadku zwraca wiadomość o kodzie 400.

Jak wspomnieliśmy wyżej, zasób ten zakłada poprawność przesyłanych wyrażeń oraz to, że argumentami, składnikami i wynikiem będą liczby całkowite. Szczegółowe i pełne działanie tego zasobu przedstawimy w rozdziale dotyczącym testów.

3.2.2 Metrics

Kolejne 3 zasoby są metrykami udostępniającymi zebrane dane dotyczące wymiany wiadomości między serwerem a klientem. Są nimi 3 zasoby:

1. GET_inputs (o ścieżce metrics/GET_inputs) - zasób, którego zadaniem jest zbieranie ilości odebranych żądań GET. Liczba żądań jest przechowywana w globalnej zmiennej GET_counter, inkrementowanej przy każdym odebraniu żądania GET.

2. PUT_inputs (o ścieżce metrics/PUT_inputs) - zasób, którego zadaniem jest zbieranie ilości odebranych żądań PUT. Zasób ten charakteryzuje się tym, że jako odpowiedź wysyła wiadomość CON, więc wymaga potwierdzenia od klienta otrzymania danych. Na jej przykładzie tez jest zrealizowane "tracenie"danych w trakcie przesyłania. Liczba żądań jest przechowywana w globalnej zmiennej PUT_counter, inkrementowanej przy każdym odebraniu żądania PUT.

3. Waiting_for_ACK (o ścieżce metrics/Waiting_for_ACK) - zasób informujący o ilości wysłanych wiadomości CON oczekujących na potwierdzenie, na podstawie licznika zawartego w strukturze `coap_session_t`. Zasób ten jest traktowany jako zasób o długim czasie dostępu, dlatego po odebraniu żądania wysyła potwierdzenie ACK, by następnie przesłać w oddzielniej wiadomości rezultat swojej pracy.

3.2.3 Zasoby w serwerze - podsumowanie

Zasoby te mają wyłączoną możliwość obserwowania. Zmienną *ct* mają ustawioną na *plain text*. W zależności od potrzeb mają zadeklarowaną reakcję na GET i PUT.

3.3 Implementacja serwera - podsumowanie

W tym rozdziale chcieliśmy przedstawić działanie naszego serwera od założeń semantycznych po rozwiązania programistyczne. Dzięki zadeklarowanej bibliotece `libcoap` tworzenie owego serwera było bardzo intuicyjne i proste.

4 Testowanie rozwiązania

4.1 Scenariusz demonstracyjny

4.1.1 Założenia projektu

Na początku chcielibyśmy przypomnieć założenia dotyczące naszego projektu. Otóż nasz projekt ma za zadanie zrealizować:

1. Obsługa wiadomości NON (GET i/lub PUT, zależnie od potrzeb dla danego zasobu). Obsługa opcji Content-Format, Uri-Path, Accept. Obsługa tokena i MID.

2. Obsługa żądań CON (GET i/lub PUT, zależnie od potrzeb dla danego zasobu) i CoAP PING. Wybrana metryka z p. 3 poniżej powinna być traktowana jako „zasób o długim czasie dostępu". W tym przypadku należy stosownie zareagować na żądanie CON, aby uniknąć retransmisji. Wysyłanie odpowiedzi CON (z retransmisją) dla wybranej metryki z p. 3 poniżej.

3. Zasób opisujący pozostałe zasoby. Ścieżka /.well-known/core. Ścieżki i atrybuty pozostałych zasobów powinny być określone przez Zespół. Obsługa GET: pobranie reprezentacji zasobu (w formacie CoRE Link Format).

4. Zbiór wyrażeń arytmetycznych zapisanych za pomocą ciągu znaków w notacji polskiej odwrotnej (RPN), np. "n n * 2 * n 3 * + 7 +" (odpowiednik 2n$\hat{2}$+3n+7). Obsługa PUT: dodanie nowego wyrażenia do zbioru (zaczynamy od zbioru pustego, a liczba elementów w zbiorze zwiększa się o 1 po każdej operacji PUT). Gdy wyczerpie się pamięć przydzielona na wyrażenia, serwer powinien odesłać odpowiedni kod błędu. Obsługa GET: pobranie wszystkich wyrażeń ze zbioru. Obsługa GET: pobranie wartości wybranego wyrażenia dla argumentu n zadanego w komponencie query w URI.

5. Trzy metryki (statystyki) opisujących wymianę wiadomości/datagramów między klientem CoAP a platformą EBSimUnoEth. Metryki powinny być zaprojektowane przez Zespół. Obsługa GET: pobranie reprezentacji metryki1. Obsługa GET: pobranie reprezentacji metryki2. Obsługa GET: pobranie reprezentacji metryki3. Jeśli Zespół ma zaimplementować obsługę żądań CON (patrz wyżej), to jedna z metryk powinna być traktowana jako „zasób o długim czasie dostępu". W tym przypadku należy stosownie zareagować na żądanie CON, aby uniknąć retransmisji. Jeśli Zespół ma zaimplementować wysyłanie odpowiedzi CON (patrz wyżej), to odpowiedzi takie powinny być generowane dla jednej z metryk. Odpowiedzi CON będziemy testować na tym zasobie.

4.1.2 Skrypt testujący

Poniżej przedstawiamy napisany przez nas skrypt w języku powłoki bash służący do testu naszego serwera:

```
coap-client -m get coap://192.168.0.143:5683/.well-known/core
echo -n "n n *" | coap-client -m put coap://192.168.0.143:5683/rpn -f
    -
coap-client -m get coap://192.168.0.143:5683/rpn?wyr=1&n=3
for i in {1...10}
```

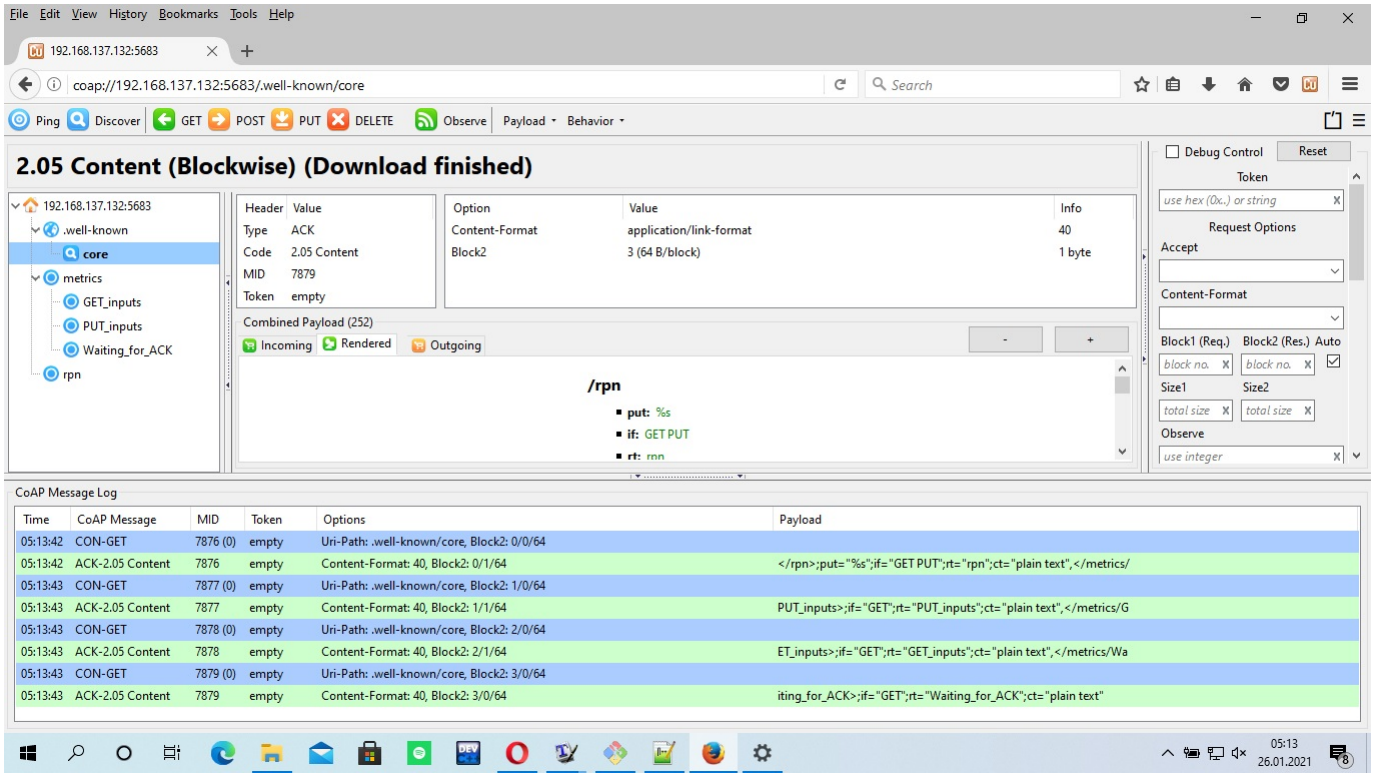
```
do
        echo -n "n n *" | coap-client -m put coap://
            192.168.0.143:5683/rpn -f-
done
coap-client -m put coap://192.168.0.143:5683/rpn?all
coap-client -m get coap://192.168.0.143:5683/metrics/GET_inputs
coap-client -m get coap://192.168.0.143:5683/metrics/PUT_inputs
coap-client -m get coap://192.168.0.143:5683/metrics/Waiting_for_ACK
```

Powyższe komendy mają na celu zaprezentować działanie naszego serwera i sprawdzenie, czy wymagania projektu są spełnione. Poszczególne kroki, które w nim realizujemy będziemy omawiać od razu przy pokazaniu wyników testów.

4.2 Wyniki testów

4.2.1 Zasób .well-known/core

W pierwszej linii wysyłamy żądanie GET do pobrania danych o wszystkich pozostałych zasobach. Realizacja tego żądania może wyglądać tak:

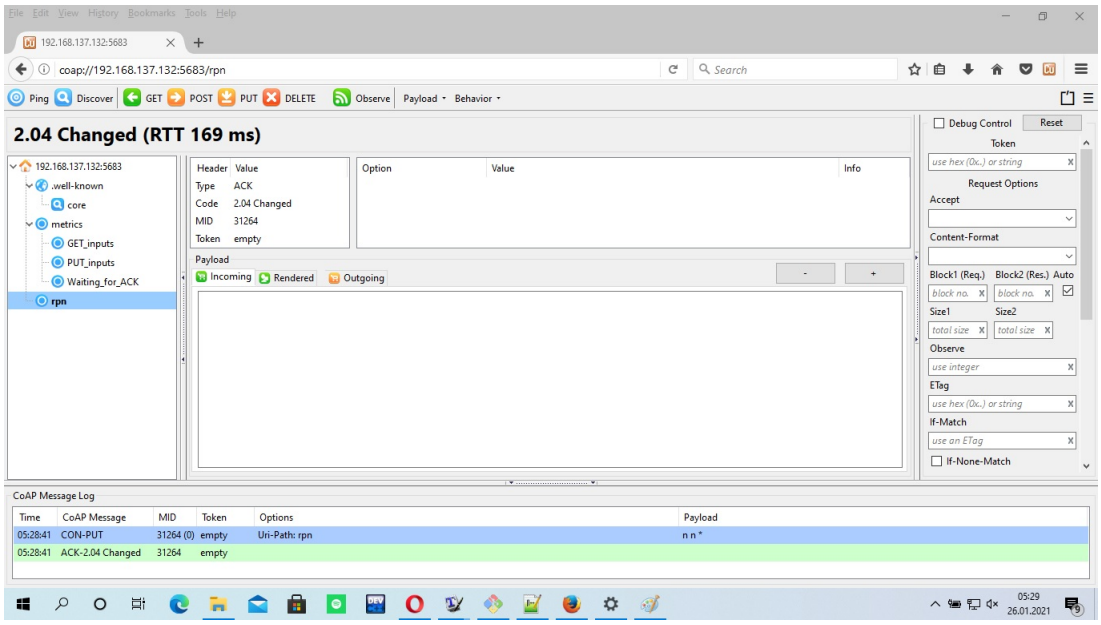


Jak widzimy, zasób ten wysyła do klienta dane o wszystkich pozostałych zasobach.

4.2.2 Zasób RPN

1. PUT

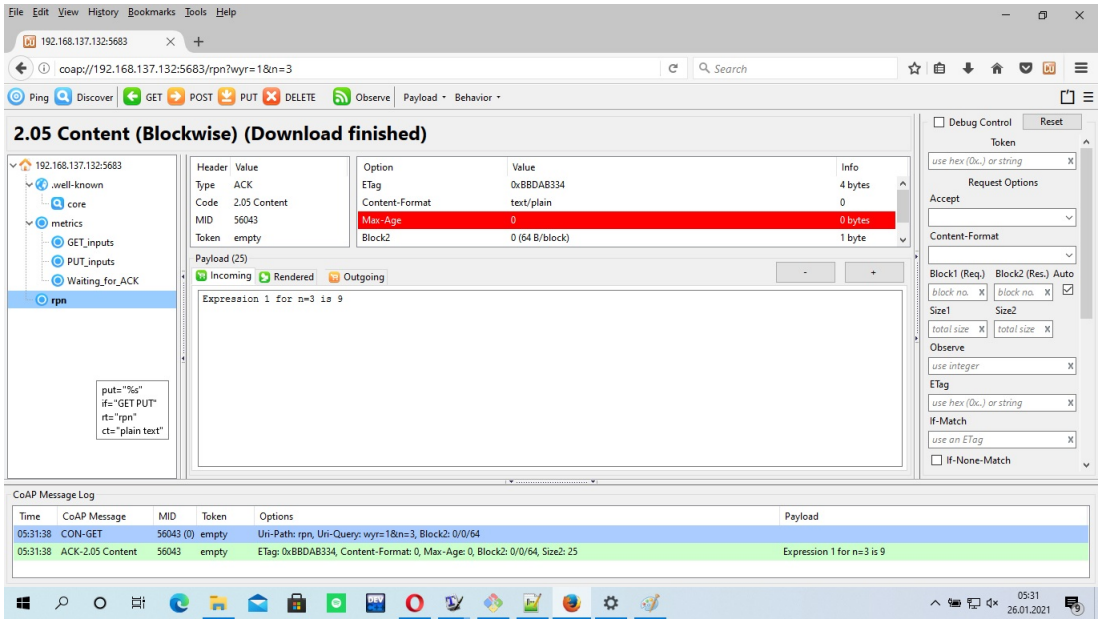
   Kolejna linia rozpoczyna testowanie zasobu *rpn*. Na początku wpisujemy dane wyrażenie. Rezultat wygląda następująco:



Dana została pobrana i otrzymaliśmy potwierdzenie.

2. GET: pobranie wyniku

   Teraz to wyrażenie, które wprowadziliśmy, obliczamy. Wynik mamy taki:

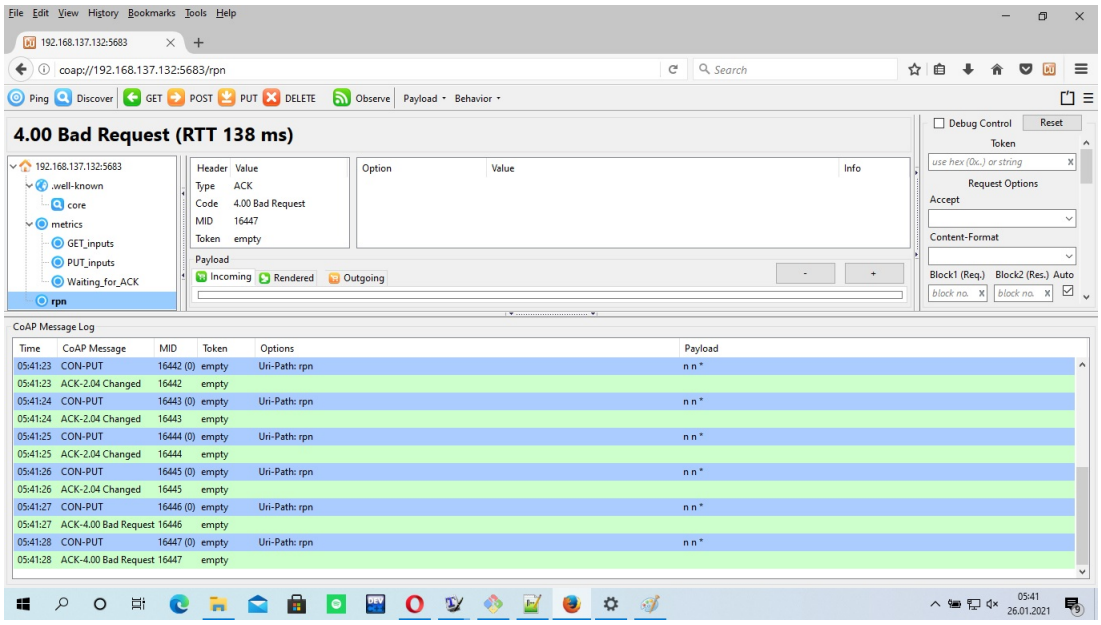


   Wynik został pobrany, jest on prawidłowy.

3. PUT: wpisanie dużej ilości wyrażeń

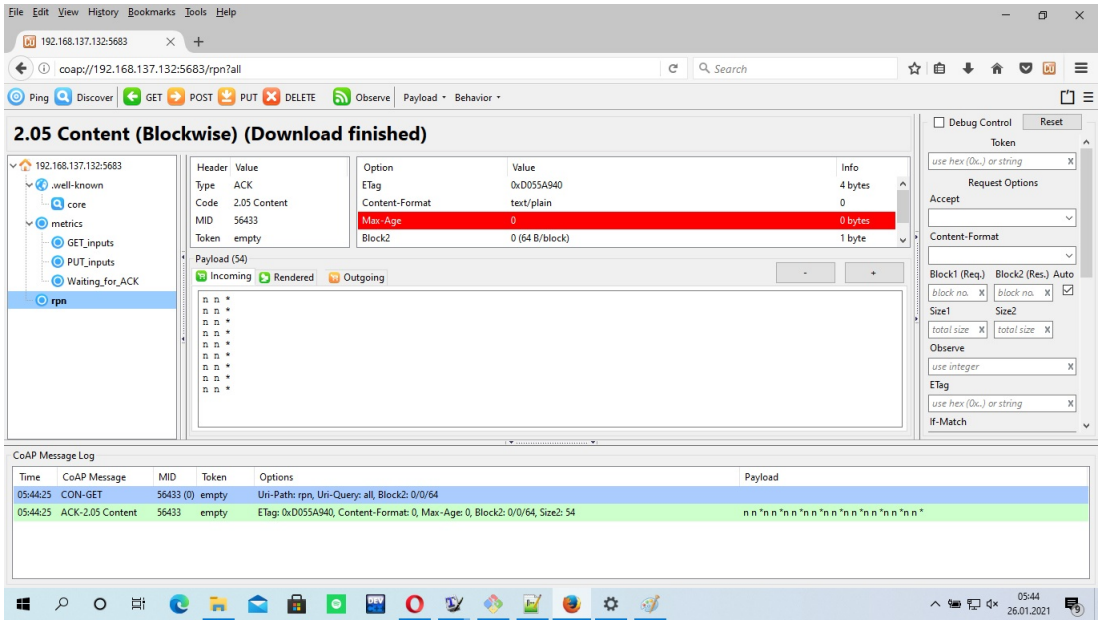
   Następnym krokiem jest pętla, która przesyła kolejne wyrażenia przekraczając limit (u nas 10 wyrażeń). Wynik tej pętli jest następujący:

   Jak widzimy, po przekroczeniu danego limitu dostajemy wiadomości zwrotnie o kodzie 400 (Bad Request). To znaczy, że nasze ograniczenie wpisania zbyt dużej ilości danych działa.



4. GET: pobranie wyrażeń

   Teraz wypiszemy nasze wyrażenia:



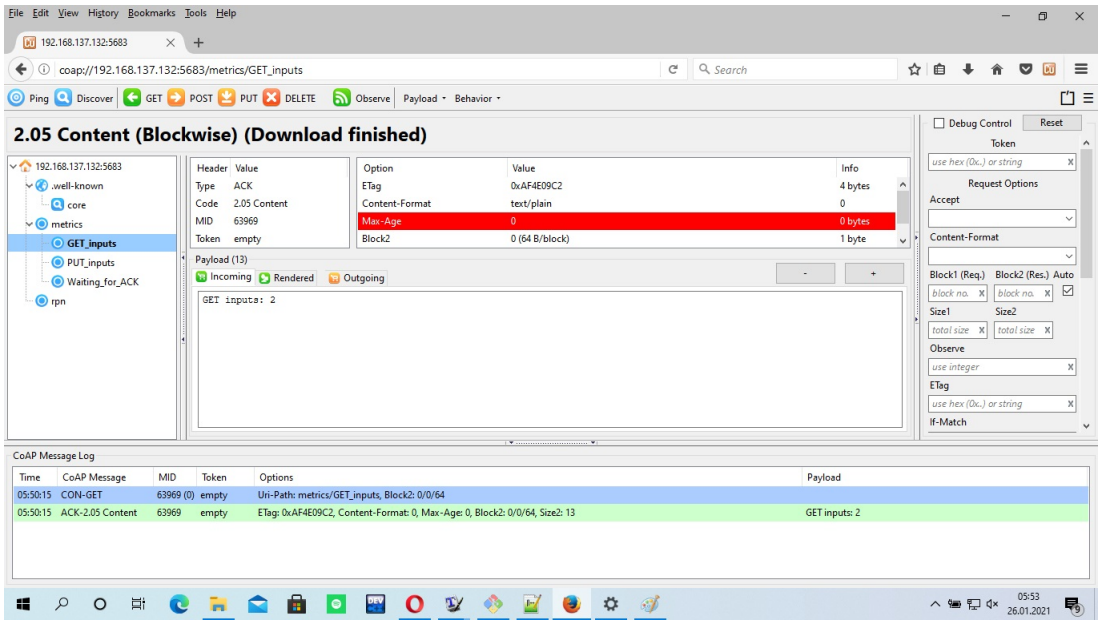
   Została nam zwrócona cała tablica wyrażeń, które poprzednio wpisaliśmy. Liczba ich jest 10.

Tym przykładem kończymy testowanie zasobu RPN.

4.2.3 Metryki

W tym podrozdziale będziemy testować metryki.

1. GET_inputs

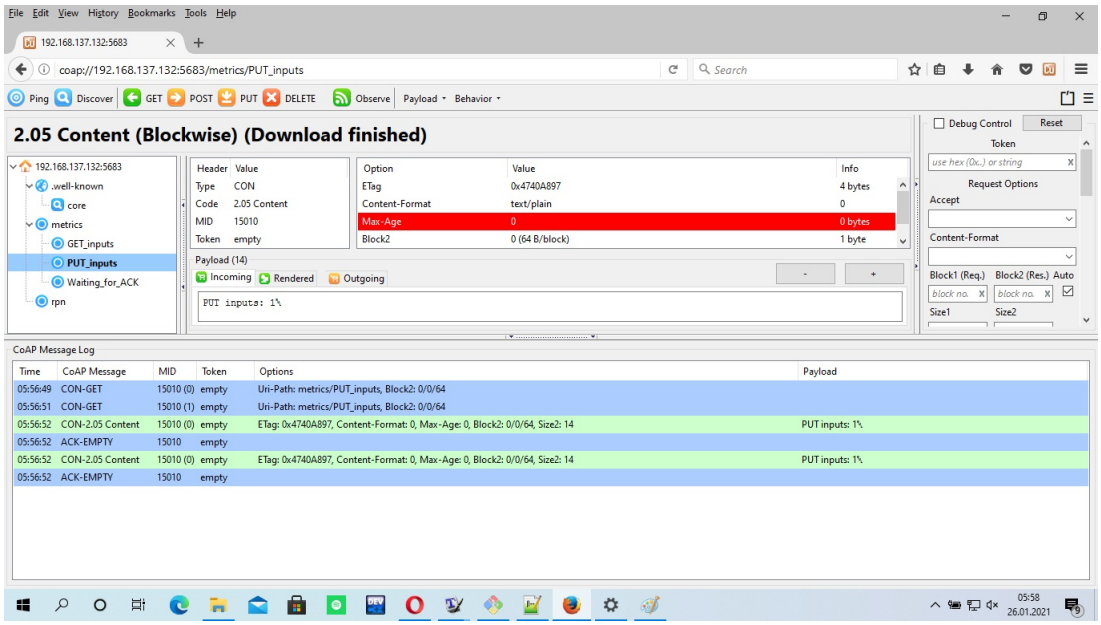


Na sam początek sprawdzimy działanie zasobu GET_inputs.

Jak widzimy, została pobrana liczba odebranych żądań GET.

2. PUT_inputs

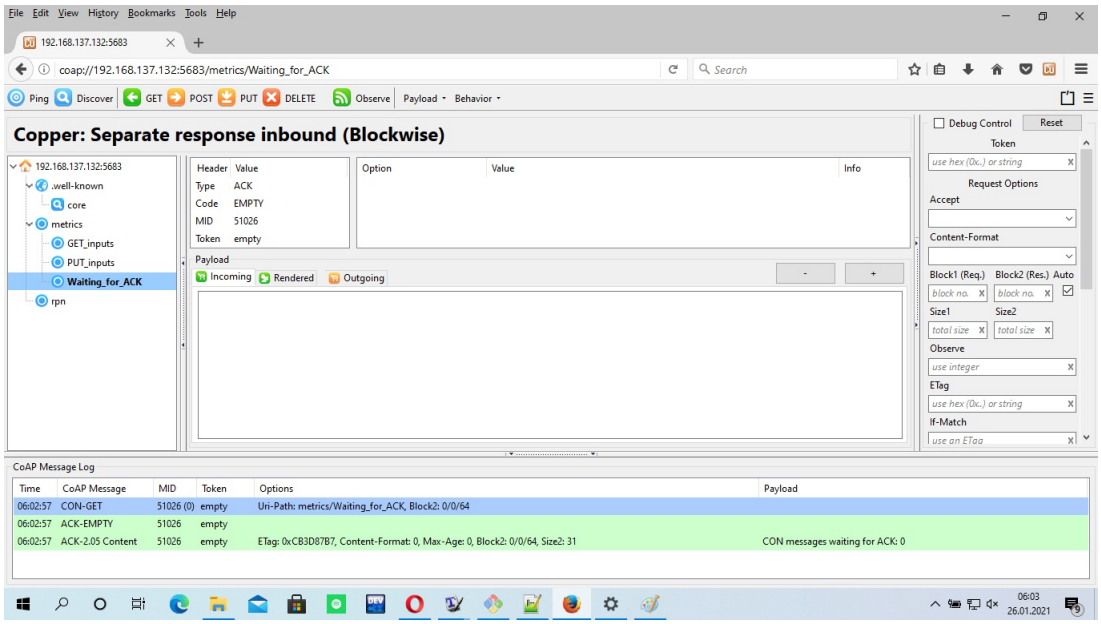
   Teraz sprawdzimy działanie zasobu GET_inputs. Należy pamiętać, że ten zasób zwraca wiadomości CON oraz jest w trakcie jego obsługi włączony tryb "gubienia"wysyłanych datagramów.



Możemy zauważyć, że klient nie dostał za pierwszym razem odpowiedzi, więc ponawia wysłanie żądania. Lecz potem serwer odpowiada 2 razy, za każdym razem dostając od klienta potwierdzenie.

3. Waiting_for_ACK

   Ostatnim zasobem, który będziemy testowac jest Waiting_for_ACK. Ten zasób natomiast jest zasobem o długim czasie dostępu, co będziemy mogli zaobserwować poniżej.



   Jak widzimy, serwer od razu odpowiada odpowiedzią ACK, by dopiero potem wysłać odpowiedź na żądanie.

4.3 Testy - podsumowanie

Dzięki poleceniom zawartym w skrypcie i scenariuszu testowania mogliśmy stwierdzić, że nasz serwer spełnia wymagania zadania.

5 Podsumowanie

Dzięki temu projektowi nauczyliśmy się nie tylko wiele na temat protokołu CoAP, ale też o strukturze typowego serwera w Internecie Rzeczy. Nasz pomysł z realizacją tego projektu nie na symulatorze Arduino, lecz na fizycznym module okazał się strzałem w dziesiątkę. Rozbudowane wymagania dotyczące całokształtu sprawiały, że projekt ten był dosć ambitny, lecz jak widać wykonalny. Dzięki temu mogliśmy też utrwalić nasze umiejętności obsługi ESP8266. Zdobyta wiedza i doświadczenie na pewno pomoga nam w dalszym rozwijaniu się w dziedzinie Internetu Rzeczy.