Politechnika Warszawska
Wydział Elektroniki I Technik Informacyjnych
Instytut Telekomunikacji

OBIEKTY INTERNETU RZECZY

Laboratorium

Ćwiczenie 2 zdalne
Temat: Komunikacja radiowa między węzłami wbudowanymi IoT.

1.Wporwadzednie

Prace z zagadnieniami komunikacji radiowej `Arduino Mini Pro` zaopatrzonej w moduł `nRF24L01+` mogą być realizowane z wykorzystaniem symulatora **EBSimMiniProRF24Network**. Jest to program, który podobnie jak **EBSimUnoEth** emuluje CPU zainstalowane w Arduino UNO, czyli `Atmega328P`.
Dla potrzeb dalszej części teksu można zdefiniować pojęcie „platforma emulowana" (`Arduino Mini Pro` zaopatrzonej w moduł `nRF24L01+`) i „platforma emulująca" (komputer osobisty na którym będzie uruchamiany **EBSimMiniProRF24Network**).
Proszę pamiętać, iż nie jest to idealna emulacja, istnieje wiele ograniczeń, wśród nich najważniejsze (poznane i wywnioskowane z budowy):
-łączność radiowa jest emulowana za pomocą rozsyłania w trybie rozsiewczym (ang. broadcast) datagramów UDP w lokalnej sieci IP (zarówno tzw. Loopback, Ethernet jak i Wifi), zatem pojedyncza instancja programu **EBSimMiniProRF24Network** może używając emulowanej łączności radiowej wymieniać się danymi z dowolną inną instancją **EBSimMiniProRF24Network** działającą na dowolnym komputerze (w tym także tym samym ale podłączonym do tej samej sieci IP), tu trzeba pamiętać, że łączność taka nie będzie przechodzić poprzez routery (np.: między interfejsem LAN i WAN) i także w niektórych konfiguracjach routerów, między siecią WiFi a Ethernet,
-mimo, że łączność jest realizowana poprzez stosunkowo wiarygodne połączenie lokalną siecią IP to tworząc oprogramowanie trzeba pamiętać, iż wiadomości mogą być gubione podczas ich transmisji takim typem łączności radiowej.

2.Instalacja środowiska

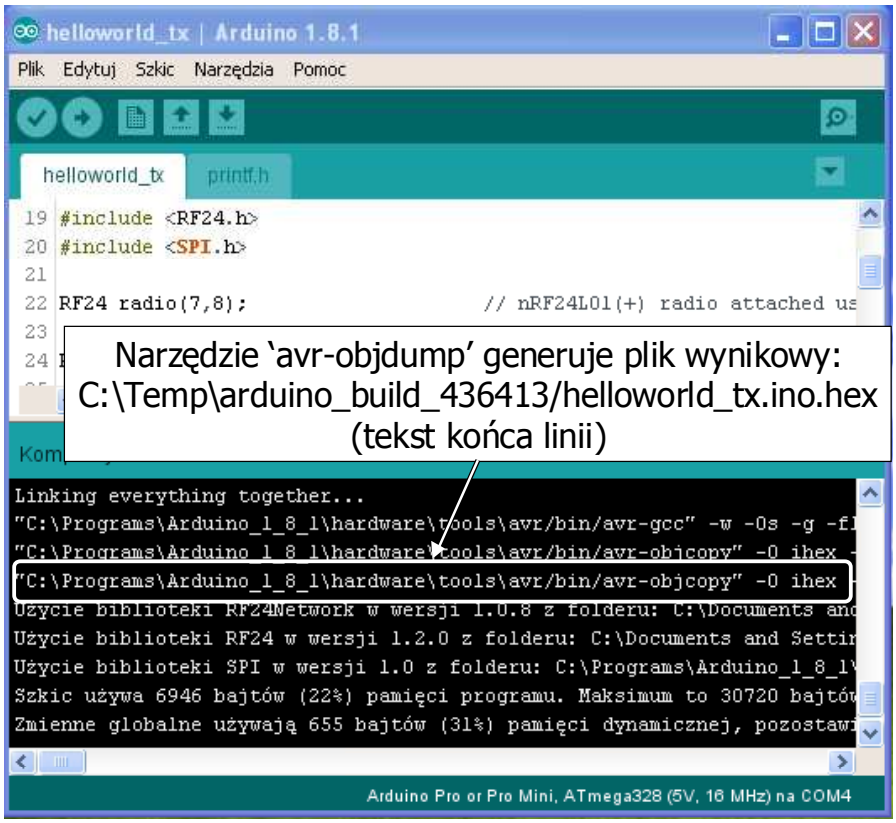
Instalacja środowiska jest niemal identyczna, jak ta przedstawiona w opisie do ćwiczenia 3 realizowanego zdalnie. Jednakże przed przystąpieniem do pracy z plikami umiejscowionymi w lokalnej kopii przydzielonego indywidualnie repozytorium GIT, należy „odświeżyć" jego zawartość. Konieczność wykonania takiej operacji wynika z faktu, iż dla potrzeb zadania 2 wykonywanego zdalnie utworzono dodatkowe pliki w tym **EBSimMiniProRF24Network.exe,** które należy pobrać z zdalnej kopii tego repozytorium. Zatem prosi się o wykonanie opisanych poniżej poleceń:

```
1.          git add .
2.          git commit –a –m „zalegle nie wypchniete pliki"
3.          git push
4.          git pull
```

Gdzie polecenia od 1 do 3 (włącznie) mogą być pominięte gdy żadne prace nie były wykonywane od ostatniego „commit'u". Natomiast na wszelki wypadek polecenie 3. może być wydane na tzw. „wszelki wypadek". Polecenie numer 4 synchronizuje lokalne repozytorium GIT ze zdalną wersją (po tym poleceniu zostanie pobrany m.in. **EBSimMiniProRF24Network**), natomiast punkty 1-3 mają zapobiec odrzuceniu polecenia 4. - odrzucenie takie może wprawić w zakłopotanie mniej doświadczonych użytkowników.

3.Tworzenie programów dla platformy EBSimMiniProRF24Network

Tworzenie oprogramowania dla platformy **EBSimMiniProRF24Network** jest identyczne jak dla platformy **EBSimUnoEth**. Dla przypomnienia należy zwrócić uwagę, że zwykle program Arduino IDE umieszcza wyniki kompilacji w katalogu **`TEMP`** (zgodnie z ustawieniami środowiskowymi systemu Windows). Przy uaktywnionej opcji raportowania etapów kompilacji (uaktywnienie wykonuje się ustawiając w `File->Preferences`: „`Show verbose output during:`” opcję `compilation`) można dokładnie ustalić miejsce, gdzie kompilator będzie umieszczał pliki wynikowe:



Należy pamiętać, iż po zamknięciu Arduino IDE program automatycznie czyści wygenerowane wcześniej wyniki kompilacji, a po ponownym uruchomieniu środowiska IDE lokalizacja plików wynikowych może ulec zmianie. Arduino IDE wyniki swojej kompilacji umieszcza w katalogu tymczasowym, będącym podkatalogiem katalogu ustawionego zmienną środowiskową `%TEMP%`, np.: `C:\Users\zsutguest\AppData\Local\Temp`.

4.Biblioteki ObirRF24 i ObirRF24Network

Na platformie **EBSimMiniProRF24Network** biblioteki `ObirRF24` i `ObirRF24Network` umożliwiają komunikację radiową. Z punktu widzenia aplikacji (czyli „sketch’y”) niemal identyczną jak podczas używania bibliotek `RF24`[1] i `RF24Network`[2] z fizycznie dołączonymi modułami `nRF24L01+`. Aby te klasy zainstalować w środowisku Arduino IDE trzeba umieścić katalogi `ObirRF24` i `ObirRF24Network` pobrane z lokalnego repozytorium GIT do lokalizacji (w przypadku użytkownika zsutguest):

`C:\Users\zsutguest\Documents\Arduino\libraries`

postępując identycznie jak w ćwiczeniu 3-zdalnym.

W kodzie tworzonym za pomocą Arduino IDE wymagane jest dodanie odpowiedniego pliku nagłówkowego:

```
#include <ObirRF24Network.h>
#include <ObirRF24.h>
```

Klasa `ObirRF24Network` emuluje właściwą komunikacje radiową, natomiast `ObirRF24` jest rozprowadzana dla zachowania zgodności z oryginalnymi bibliotekami. Podobnie dla zachowania zgodności można w swoim szkicu umieścić zapis:

```
RF24 radio(7, 8);
```

Ten fragment nie zmienia nic w oprogramowaniu wynikowym, a co więcej system nie sprawdza czy, aby odpowiednie zasoby (tutaj piny 7 i 8) są wolne i można je tak wykorzystać.
Implementacja klasy ObirRF24Network jest uproszczona do maksimum i wiele z metod tej klasy mimo, iż istnieje to nie posiada żadnej implementacji. Dla dalszych prac sugeruje się korzystać wyłącznie z metod tej klasy:

```
void begin(uint8_t _channel, uint16_t _node_address);
uint8_t update(void);
bool available(void);
uint16_t read(ObirRF24NetworkHeader& header, void* message, uint16_t maxlen);
bool write(ObirRF24NetworkHeader& header, const void* message, uint16_t len);
```

Do klasy tej dodano dwie nowe metody:

`uint8_t error(void);` - pobranie kodu ostatnio odnotowanego błędu (0x01 –adres odbiorcy poza schematem adresacji np.: 0xFFFF, 0x02 – pakiet za krótki (zalecane odczekanie, bo może nie został odebrany cały), 0x04 – pakiet odebrano ale nie jest dla nas,

`uint16_t get_radio_packet_reciever_address(void);` - gdy pakiet został odebrany aplikacja może dowiedzieć się o adres nadawcy tego pakietu.

Podobnie jak podczas pracy z fizycznym modułem `nRF24L01+`, tworząc obiekt klasy `ObirRF24Network` należy wskazać obiekt klasy współpracujący bezpośrednio z układem radiowym (zdefiniowany powyżej):

```
RF24Network network(radio);
```

W funkcji `setup()` aplikacji dla zachowania zgodności z fizycznym modułem `nRF24L01+` można zainicjować interfejs SPI oraz obiektu klasy `RF24`:

```
SPI.begin();
radio.begin();
```

Zakłada się, że podczas prac nad ćwiczeniem 2 realizowanym zdalnie, współpracować będą dwa węzły – oba emulowane przez tę samą „platformę emulującą”, czyli komputer osobisty na którym będzie uruchamiany **EBSimMiniProRF24Network**. Jednakże podczas korzystaniem z komunikacji radiowej, każda z „platform emulowanych” będzie musiała wykonywać aplikację (szkic) komunikującą się z węzłem-partnerem z zachowaniem odpowiednio dobranych identyfikatorów węzłów.
Zakładając, że węzeł będzie miał ustalony identyfikator jako `THIS_NODE_ID` - liczbowo np.: `0x00`, to węzeł-partner musi posiadać swój identyfikator także o nazwie `THIS_NODE_ID` ale liczbowo równy np.: `0x01`. Dodatkowo oba węzły muszą używać wspólny kanał radiowy (tu oznaczany jako `OUR_CHANNEL`). Można to uczynić inicjalizując obiekt `network`:

```
network.begin(OUR_CHANNEL, THIS_NODE_ID);
```

Pracując z fizycznymi modułami `nRF24L01+` bardzo ważne jest, aby pamiętać o specyfice nadawania unikatowych adresów węzłom – w emulowanym środowisku także można zachować tę regułę. W sieci można znaleźć dokument opisujący to zagadnienie[3]. Specyfikuje on metodykę postępowania przy przydzielaniu odpowiednich wartości tym adresom. Dla prostych dwu-

węzłowych topologii można dla uproszczenia przyjąć, iż jeden z węzłów będzie miał adres 0, a drugi 1.
Aby zachować właściwe przypisanie identyfikatorów radiowych węzłów, uruchomienie dwóch węzłów można zrealizować poprzez wydanie poniższych poleceń w oknie terminala (CMD) polecenia – emulacja węzła serwerowgo:

```
cd EBSimMiniProRF24Network\bin-dist\win10\cygwin
PATH=./dll-win10;%PATH%%
EBSimMiniProRF24Network.exe ObirRF24Network_helloworld_rx_20200505.ino.hex
```

Oraz w osobnym oknie terminala - emulacja węzła klienckiego:

```
cd EBSimMiniProRF24Network\bin-dist\win10\cygwin
PATH=./dll-win10;%PATH%%
EBSimMiniProRF24Network.exe ObirRF24Network_helloworld_tx_20200505.ino.hex
```

Dzięki powyższemu oba węzły uruchomią emulacje z innym „wsadem" a przez zagwarantowanie iż szkic `ObirRF24Network_helloworld_rx_20200505.ino` zadeklarował zmienną „`this_node`" wartością 0 (`THIS_NODE_ID`) a drugi `ObirRF24Network_helloworld_tx_20200505.ino` na 1, oba emulatory nie będą sobie przeszkadzać podczas komunikacji radiowej.
Treść przesyłana między węzłami używającymi modułów `nRF24L01+` i klasy `RF24Network` może być nie większa niż 132B zakładając, że klasa ta dokona wewnętrznie fragmentacji lub 32B gdy ta fragmentacja zostanie wyłączona. Implementacja emulatora **EBSimMiniProRF24Network** zakłada jednak, że najdłuższy pakiet może mieć długość 120B i taką wielkość wiadomości jest wstanie przesłać `ObirRF24Network` i nie ma dzielenia wiadomości na mniejsze fragmenty.
Podążając zatem za przykładami przygotowanymi dla klasy `RF24Network`[4][5], które będą także działały z użyciem klasy `ObirRF24Network` do przesyłania wiadomości warstwy aplikacji można zdefiniować strukturę:

```
struct payload_t {
  unsigned long ms;          // stempel czasowy
  unsigned long counter;     // nr seryjny wysylanej wiadomosci
};
```

Przed wysłaniem nowej wiadomości należy: a)wypełnić odpowiednie pola tej struktury, b)utworzyć nagłówek wiadomości (obiekt klasy `ObirRF24NetworkHeader`) określając jej adresata, c)przekazać za pomocą funkcji `write()` wiadomość do wysłania:

```
struct payload_t payload;
payload.ms =      ...
payload.counter = ...
ObirRF24NetworkHeader header(PEER_NODE_ID);      //tu podajemy id adresata
network.write(header, &payload, sizeof(payload));
```

W implementacji klasy `RF24Network` metoda `write()` zwraca informację, czy wysyłanie powiodło się. Domyślnie włączony mechanizm potwierdzeń, wspierany przez układ `nRF24L01+`, sprawia, że metoda `write()` na czas oczekiwania na potwierdzenie zostanie zablokowana. Maksymalnie czas ten może wynosić do 75ms (w implementacji decyduje o tym wewnętrzna zmienna: `routeTimeout`). Jeżeli po tym czasie nie zostanie otrzymane potwierdzenie metoda `write()` zwróci `FALSE`, w przeciwnym przypadku, tj. gdy w tym czasie potwierdzenie nadejdzie, metoda `write()` zwróci `TRUE`. W przypadku wysyłania wiadomości - z użyciem fizycznego modułu - których payload jest zbyt długi, biblioteka będzie dzielić payload na fragmenty. Warto pamiętać, że fragmentacja może dodatkowo zwielokrotnić czas blokowania funkcji `write()`.
Tu pojawiają się kolejne różnice w implementacji `ObirRF24Network`. Po pierwsze nie wspiera ona mechanizmu potwierdzeń i po wywołaniu metody `write()` zwróci niemal natychmiast `TRUE`. Drugą

różnicą w implementacji jest brak mechanizmu fragmentacji, każdy pakiet dłuższy od 120B nie zostanie wysłany, natomiast każdy pakiet krótszy od tej wartości zostanie wysłany za pomocą łączności radiowej emulowanej za pomocą lokalnej sieci IP.
Korzystając z implementacji biblioteki `RF24Network`, aby nadawca wiadomości mógł otrzymać potwierdzenie, adresat tej wiadomości musi jak najszybciej potwierdzić odebranie takiej wiadomości. Biblioteka `RF24Network` realizuje tą operację, ale aby to mogło być możliwe aplikacja powinna cyklicznie wywoływać metodę biblioteczną:

```
network.update();
```

Metoda `update()` powinna być możliwie często wywoływana; z tego powodu autorzy bibliotek `RF24`/`RF24Network` zalecają, aby pozostały kod funkcji `loop()` był wykonywany jak najkrócej. W przypadku użycia klasy `ObirRF24Network` wywoływanie metody `update()` jest zbędne, ale dla zachowania zgodności z oryginalną biblioteką można je pozostawić.
W przypadku, gdy węzeł chce odebrać wiadomość przesyłaną drogą radiową, powinien cyklicznie sprawdzać stan obiektu `network` (klasy `RF24Network`), za pomocą funkcji `available()` oraz, gdy zorientuje się, że są jakieś dane, odczytać treść takich danych:

```
void loop(){
      // handle radio module
      network.update();
      while(network.available()){
            network.read(header, &payload, sizeof(payload));
            ...                              //przetwarzanie treści wiadomości
      }
      //obsługa innych zadań
      ...
}
```

Implementacja klasy `ObirRF24Network` tym zakresie pozwala na identyczne postępowanie.

5. Komunikacja radiowa z wykorzystaniem podejścia klient-serwer

Jak łatwo zauważyć konstrukcja `struct payload_t` jest mało elastyczna. Stosując tę strukturę trzeba zadbać aby obie strony posiadały identyczną jej definicje (co często wprowadza wiele błędów podczas programowania). Generalnie jednak jest to bardziej forma przekazywania wartości niż zlecania przez jedną ze stron pewnych zadań do wykonania przez drugą stronę a przekazywanie drogą radiową całego stanu węzła z którym współpracujemy jest operacją nieco nie efektywną. Dzieje się tak dlatego, że stany nie wszystkich zasobów węzła z którym wymieniamy się danymi muszą zmieniać się z taką samą częstością, więc cześć treści przekazywana jest nadmiarowo. Naturalne jest zatem podejście w którym przekazywane będą na żądanie tylko wybrane pola takiej struktury.
Aby jednak takie operacje zrealizować, konieczne byłoby dodanie specjalnego mechanizmu dzięki któremu jeden z węzłów mógłby wskazywać jakie pole chciałby sobie odświeżyć. Widać tu zatem typową konstrukcję Klient-Serwer. Jeden z węzłów (klient) pragnie aby drugi węzeł coś dla niego wykonał (serwer), np.: odświeżył jego wiedzę o stanie jakiegoś pola z wewnętrznych struktur opisujących jego stan.
Tego typu zadanie można realizować na różne sposoby. W ramach prac badawczych w zespole MEAG-ITPW[6] powstał specjalny protokół `nanoRPC`, który wraz z narzędziem do automatycznej generacji oprogramowania[7] powyższe zadanie bardzo mocno upraszcza. Za pomocą tego generatora (dostępnego przez stronę WWW[8]) można automatycznie wygenerować oprogramowanie które łatwo zintegrować z resztą oprogramowania tworzonego także z wykorzystaniem Arduino IDE.
Pierwszym krokiem jaki należy uczynić to dla zdefiniowanej listy usług jakie serwer ma realizować trzeba utworzyć plik XML opisujący ją formalnie:

`<node>`	
`<name>wezel_1</name>`	Dowolna*) nazwa węzła
`<service>`	Definicja pierwszej usługi
`<def>nanoRpcGetTime</def>`	Unikatowa dowolna*) nazwa usługi
`<result>uint32_t</result>`	Typ rezultatu **) ***)
`</service>`	
`<service>`	Definicja drugiej usługi
`<def>nanoRpcGetCounter</def>`	Unikatowa dowolna*) nazwa usługi
`<result>uint32_t</result>`	Typ rezultatu **) ***)
`</service>`	
`<service>`	Definicja trzeciej usługi
`<def>nanoRpcSetTime</def>`	Unikatowa dowolna*) nazwa usługi
`<args>`	Lista argumentów podanej usługi
`<type>uint32_t</type>`	
`</args>`	Koniec listy argumentów tej usługi
`<result>uint32_t</result>`	Typ rezultatu **) ***)
`</service>`	
`</node>`	

*)Nazwa dowolna ale bez spacji i polskich znaków diakrytycznych

**)Dozwolone typy to: uint8_t, uint16_t, uint32_t, int8_t, int16_t, int32_t

**)W obecnej wersji narzędzia, sugerowane jest aby każda usługa zwracała jakiś rezultat (np.: przynajmniej kod błędu wykonania danej usługi, nawet gdyby zawsze było to zero).

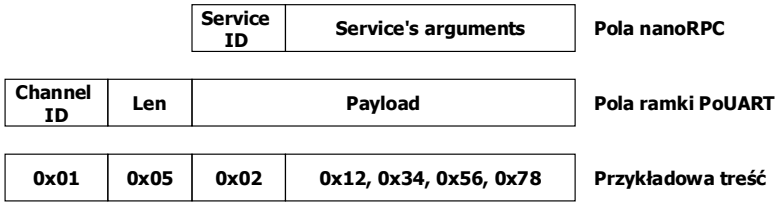
Na podstawie powyższej treści można powiedzieć, że dostępne staną się usługi `nanoRPC` realizowane przez następujące funkcje:

```
uint32_t nanoRpcGetTime(void);
uint32_t nanoRpcGetCounter(void);
uint32_t nanoRpcSetTime(uint32_t);
```

Powyżej opisany plik XML trzeba „wrzucić” przez stronę WWW generatorowi `nanoRPC`, po czym w wyniku pracy generatora otrzymamy plik z rozszerzeniem ZIP. Pliki wchodzące w skład otrzymanego pliku ZIP należy skopiować do katalogów gdzie będą przechowywane szkice klienta i serwera, z zachowaniem lokacji:

`Szkic serwera`	`Szkic klienta`
`msg_interpreter.h`	`msg_gen.h`
`msg_ret_gen.h`	`msg_ret_interpreter.h`
`nanoRPC_iface.h`	`nanoRPC_iface.h` `kopia pliku identyczna jak dla serwera`

Pojawić się mogą pytania: a jak połączyć logikę klasy `ObirRF24Network` z wygenerowanymi plikami, jak używać usługi realizowane przez `nanoRPC` (strona klienta) i jak je zaimplementować (strona serwera). Zacznijmy jednak od domyślnego formatu danych wymienianych przez protokół `nanoRPC`. Bazuje on na tzw. ramkach `PoUART`. W ramce tej umieszczane są odpowiednie informacje przenoszone przez ten protokół, co dla usługi `nanoRpcSetTime` pokazuje rysunek:



Implementacja protokołu nanoRPC z punktu widzenia wymiany danych zakłada, że dostarczone będą jej odpowiednie usługi:
a)funkcji umożliwiającej nanoRPC wysłanie ramki PoUART do współpracującego węzła:

```
      void sendPoUART(uint8_t channel, uint8_t payload_len, uint8_t *p)
```

z założeniem, że: channel - to numer wirtualnego kanału używanego podczas stosowania ramek PoUART (ten kanał nie ma nic wspólnego z numerek kanału radiowego stosowanego w implementacji RF24Network czy ObirRF24Network), argument: payload_len - przekaże informację o długości przenoszonych przez ramkę PoUART danych które będą wskazane przez zmienną 'p',
b)zestaw funkcji konwersji tzw. „końcówkowości" (w poniższych przykładach zakłada się, że oba węzły muszą działać z tą samą końcówkowością):
MHTONL(), MNTOHL(), MHTONS(), MNTOHS(),
c)implementacji funkcji obsługi błędów wykrytych przez implementacje protokółu nanoRPC podczas swojej pracy:

```
      void NANORPC_FRAME_ERROR(uint8_t error_val)
```

tutaj przekazany argument to numeryczny kod błędu – szczegóły podane są w implementacji nanoRPC.
Natomiast gdy niższe warstwy komunikacyjne węzła odbiorą ramkę PoUART, przekażą ją za pomocą wywołania:

```
      void recvPoUART(uint8_t channel, uint8_t payload_len, uint8_t *p)
```

Implementacja takich funkcji może wyglądać następująco:

```
#define MAX_OBIRRF24NETWORK_PAYLOAD (120)
void recvPoUART(uint8_t c, uint8_t l, uint8_t *r); //tu tylko prototyp(!)
uint32_t MHTONL(uint32_t p){return p;}
uint32_t MNTOHL(uint32_t p){return p;}
uint16_t MHTONS(uint16_t p){return p;}
uint16_t MNTOHS(uint16_t p){return p;}
void NANORPC_FRAME_ERROR(uint8_t error_val){
    //implementacja dowolna, zależna od konstrukcji węzła
}
void sendPoUART(uint8_t channel, uint8_t payload_len, uint8_t *p){
    uint8_t payload[2+payload_len];
    ObirRF24NetworkHeader header(other_node);
    payload[0]=channel;
    payload[1]=payload_len;
    memcpy(&(payload[2]), p, payload_len);
    bool ok=network.write(header, payload, 2+payload_len);
    if(!ok)
        Serial.println(F("failed."));
}
```

Następnie (tutaj kolejność w kodzie ma znaczenie) konieczne jest załączenie wygenerowanych plików – w kodzie serwera:

```
#include "msg_interpreter.h"
#include "msg_ret_gen.h"
#include "nanoRPC_iface.h"
```

oraz w kodzie klienta:

```
#include "msg_gen.h"
#include "msg_ret_interpreter.h"
#include "nanoRPC_iface.h"          //ten sam plik co użyty w kodzie serwera
```

A następnie trzeba w kodzie serwera zdefiniować implementację realizacji właściwych usług, np.:

```
void nanoRpcGetTimeSrv(){
      //... zależna od konstrukcji węzła
      nanoRpcGetTimeRet(...);      //zwrot rezultatu działania tej usługi
}
void nanoRpcGetCounterSrv(){
      //... zależna od konstrukcji węzła
      nanoRpcGetCounterRet(...);   //zwrot rezultatu działania tej usługi
}
void nanoRpcSetTimeSrv(uint32_t a1){
      //... zależna od konstrukcji węzła
      nanoRpcSetTimeRet(...);      //zwrot rezultatu działania tej usługi
}
```

Odtąd w kodzie klienta będzie można używać usługi:

```
void nanoRpcGetTime(void);
void nanoRpcGetCounter(void);
void nanoRpcSetTime(uint32_t a);
```

Zastanawiać może dlaczego mimo zdefiniowania w pliku XML funkcji które miały zwracać wyniki typu `uint32_t`, żadna z powyższych nic nie zwraca. Wynika to z przyjętego z punktu widzenia klienta rozdzielenia realizacji usługi na dwie części: fazy zlecenia wykonania usługi i fazy zwrotu wyniku działania danej usługi. Zatem dla powyższego przykładu konieczne jest zaimplementowanie zestawu tzw. funkcji zwrotnych po stronie klienta – to faktycznie one otrzymają wynik wywołania danej usługi:

```
void nanoRpcGetTimeSrvRet(uint32_t a){
      //...
}
void nanoRpcGetCounterSrvRet(uint32_t a){
      //...
}
void nanoRpcSetTimeSrvRet(uint32_t a){
      //...
}
```

Takie rozdzielenie wydaje się nieco dziwne jest jednak niezbędne gdyż domyślnie aplikacje w Arduino API nie wspierają mechanizmów wielozadaniowości a do tego nierozważnie byłoby blokować działanie klienta na cały czas wykonania usługi przez serwer oraz transferu danych między klientem i serwerem.
Podsumowując logiczny tok współdziałania obu węzłów jest następujący:
a)po stronie klient mieliśmy zadeklarowaną usługę: `uint32_t nanoRpcSetTime(uint32_t)` dla jej wywołania klient jednak w kodzie aplikacji musi wywołać funkcję: `void nanoRpcSetTime(uint32_t a)`,
b)serwer po otrzymaniu odpowiednich wiadomości dzięki implementacji protokołu `nanoRPC` wywoła funkcję implementacji usługi (to tu twórca szkicu może połączyć usługę z fizycznymi elementami węzła): `void nanoRpcSetTimeSrv(uint32_t a1)`, a w ramach implementacji tej funkcji twórca szkicu przed jej końcem powinien wywołać funkcję: `void nanoRpcSetTimeRet(uint32_t a)`, przekazując w argumencie ‘a’ zwracaną do klienta wartość,
c)na zakończenie po stronie klienta po otrzymaniu odpowiednich wiadomości protokoł `nanoRPC` wywoła funkcję zwrotną `void nanoRpcSetTimeSrvRet(uint32_t a)`, przekazując warstwie aplikacji finalny wynik działania usługi `nanoRpcSetTime`.
Tworząc odpowiedni kod może powstać pytanie: jak zaimplementować kod głównych części szkicu. Ta część jest niemal neutralna dla obu elementów tego systemu: serwera i klienta i wyglądają one niemal identycznie (różnicę zawarto w komentarzu):

```
void setup(void){
  radio.begin();
  network.begin(/*channel*/ 90, /*node address*/ this_node);
}
```

```
void loop(void){
  network.update();

  //w kodzie klienta tutaj można umieścić jakieś jedno z wywołań:
  //nanoRpcGetTime(), nanoRpcGetCounter(), nanoRpcSetTime()
  //w kodzie serwera – nic co jest związane z nanoRPC

  //część wspólna dla obu węzłów
  while(network.available()){
    ObirRF24NetworkHeader header;
    uint8_t payload[MAX_OBIRRF24NETWORK_PAYLOAD];
    if(network.read(header, &payload, sizeof(payload))>0){
       recvPoUART(payload[0], payload[1], (uint8_t*)(&(payload[2])));
} } }
```

Należy zauważyć, iż podane powyżej fragmenty kodu nie zawierają wielu, niekiedy kluczowych elementów sprawdzenia pomyślnego wywołania funkcji i metod.

Linki:

1. http://tmrh20.github.io/RF24/
2. http://tmrh20.github.io/RF24Network/
3. http://tmrh20.github.io/RF24Network/Addressing.html
4. http://tmrh20.github.io/RF24Network/helloworld_rx_8ino-example.html
5. http://tmrh20.github.io/RF24Network/helloworld_tx_8ino-example.html
6. https://meag.tele.pw.edu.pl
7. Pruszkowski2015MaszynoweIot_wyciag_dla_przedmiotu_OBIR.pdf
8. https://equ3.tele.pw.edu.pl/nanoRPC