

Politechnika Warszawska

PSIR: Laboratorium 2

Podstawy komunikacji sieciowej z wykorzystaniem platformy i API
Arduino

Kazimierz Kochan 303704

Monika Lewandowska 303707

2022-04-13

Wstęp.....	2
Zadanie 1.....	2
Zadanie 2.....	3

Wstęp

Laboratorium składa się z dwóch zadań pozwalających w sposób praktyczny przyswoić wiedzę z wykładu dotyczącą tworzenia i uruchamiania oprogramowania sieciowego w systemie Linux bazującego na socketach. Oba ćwiczenia laboratoryjne wykonaliśmy w parze stosując się poleceń z instrukcji – Kochan_Kazimierz_Cyryl.pdf. Plik znajdują się w repozytorium GIT Kazimierz Kochana w folderze lab2 oraz odpowiednich folderach zad1 i zad2.

Zadanie 1

Zakładając architekturę klient-serwer, napisać dla platformy Arduino-emulowanej, odpowiednie oprogramowanie. Oprogramowanie to ma cyklicznie co 3070 ms (zgodnie z działaniem `ZsutMillis()`) raportować stan GPIO o oznaczeniu Z0, a do komunikacji należy wykorzystać należy protokół UDP na porcie 12370 (port serwera). Elementem odbierającym dane w tym zadaniu niech będzie narzędzie netcat, zaproponuj odpowiednią treść pliku `infile.txt` dowodzącą, że system działa, a całość testuj z wykorzystaniem emulatora Arduino o nazwie `EBSimUnoEthCurses`.

Zgodnie z poleceniem zaimplementowaliśmy serwer dla emulatora Arduino, którego zadaniem jest cykliczne wysyłanie raportu o stanie pinu Z0. Kod programu `server.cpp` zamieściliśmy w repozytorium w dedykowanym katalogu.

Rolę klienta pełni narzędzie netcat, a w celu przetestowania poprawności naszego rozwiązania przygotowaliśmy plik `infile.txt`. Ustawiamy w nim zmieniające się wartości na pinie Z0, które serwer odczytuje i wysyła.

```
psir@psir21z:~/EBSimUnoEthCurses_workspace$ cat infile.txt
#0pis zasobow
+ qTemperature,quantity,Z0
#test nr.1
: 7500000, qTemperature, 0
: 20000000, qTemperature, 10
: 25000000, qTemperature, 100
: 30000000, qTemperature, 1000
: 35000000, qTemperature, 22
: 40000000, qTemperature, 13
: 50000000, qTemperature, 598
```

Rysunek 1 `infile.txt`

```
GPIO
Z0: 0x0256 (00598)      D0: 1      D7: 1
Z1: 0x03ff (01023)      D1: 1      D8: 1
Z2: 0x03ff (01023)      D2: 1      D9: 1
Z3: 0x03ff (01023)      D3: 1      D10: 1
Z4: 0x03ff (01023)      D4: 1      D11: 1
Z5: 0x03ff (01023)      D5: 1      D12: 1
                        D6: 1      D13: 1

UART
ZSUT eth UDP server init...
src/main.cpp
Apr 20 2022, 23:22:11
IP address: 192.168.56.102

Send: 0.
Using 2 bytes: 0, 0

Send: 1000.
Using 2 bytes: 232, 3

Send: 598.
Using 2 bytes: 86, 2
```

Rysunek 2 Symulacja

Jak widać, nie wszystkie wartości zostały przesłane. Jest to zachowanie normalne i celowe. Pokazuje ono, że datagramy są wysyłane co określony interwał: 3070ms, a to co dzieje się w czasie pomiędzy wysyłaniem danych nie będzie widoczne dla klienta.

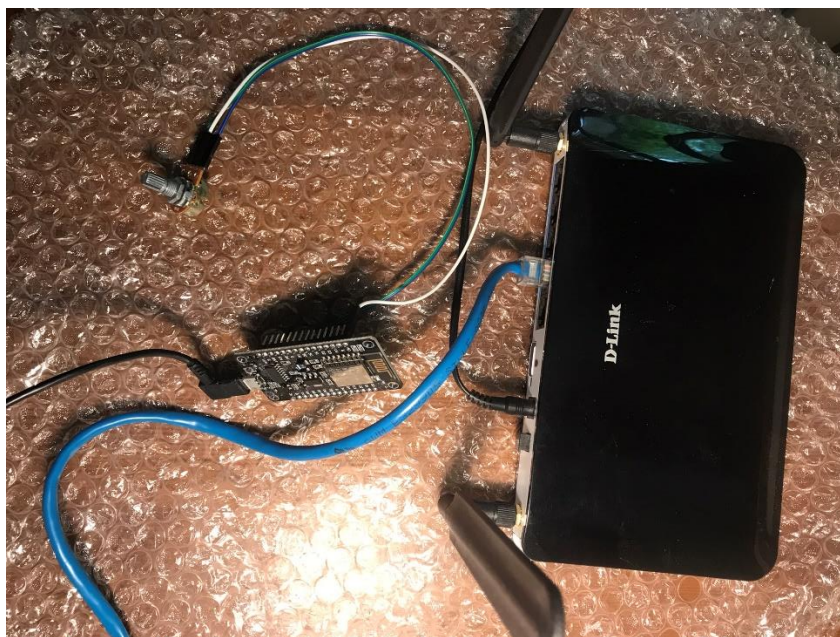
Zadanie 2

Zakładając architekturę w której mamy Arduino-emulowane (działające jako klient) i Arduino-sprzętowe (działające jako serwer) współpracują ze sobą w taki sposób, że klient korzysta z pracy serwera. Kod serwera ma być napisany z wykorzystaniem Arduino IDE lub Platformio a emulowany przez EBSimUnoEthCurses, kod klienta należy utworzyć także z wykorzystaniem Arduino IDE lub Platformio ale ma pracować na natywnej platformie Arduino UNO. Kod serwera na porcie 8276 ma przekazywać po swoim uruchomieniu wiadomość protokołu UDP zgłaszając w ten sposób gotowość do pracy (odpowiednik wiadomości HELLO), aby następnie zgodnie z żądaniami otrzymywanymi od klienta odsyłać mu raporty dotyczące napięcia na wejściu A2 do którego dołączony jest potencjometr zgodnie z opisem zamieszczonym w dokumencie wprowadzającym (rys.6, str. 6). Żądania klient ma wysyłać serwerowi co 3380 ms. Klient odebrawszy każdy z raportów, ma demonstrować ich treść z wykorzystaniem swojej konsoli szeregowej w sposób umożliwiający diagnozowanie czasów kiedy taki raport odebrano.

Adres MAC, jaki został przydzielony dla naszego zespołu to: b8:27:eb:fb:eb:80.

Środowisko

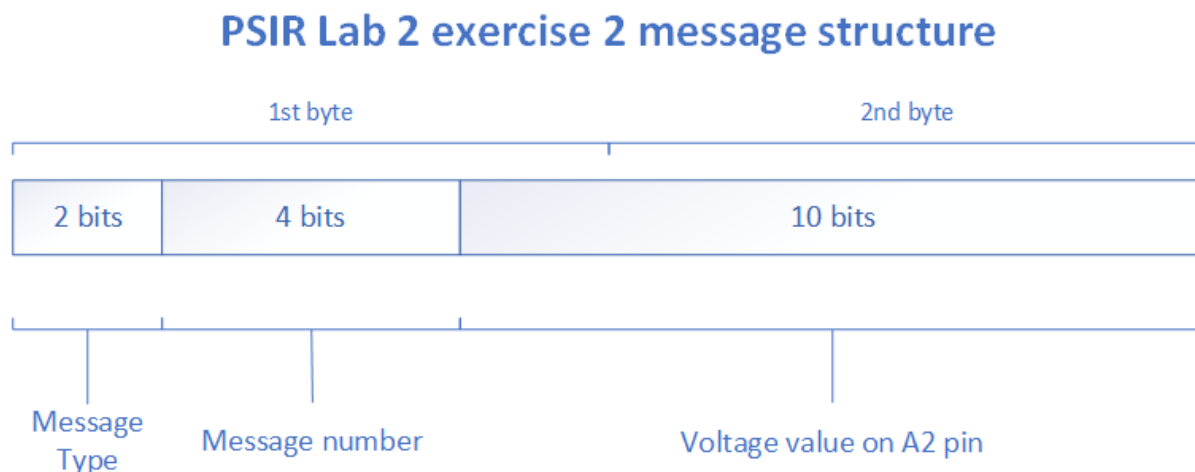
Finalną wersję rozwiązania przygotowaliśmy w domu korzystając z mikrochipu ESP8266 połączonego za pomocą protokołu WiFi do sieci bezprzewodowej. Komputer, na którym uruchomiona była maszyna wirtualna z emulatorem została podłączona do routera za pomocą kabla Ethernetowego. Z powodu mniejszej liczby wejść analogowych zamiast wejścia A2 z instrukcji skorzystaliśmy z jedynego wejścia A0.



Rysunek 3 Architektura dokończonego w domu laboratorium

Opis struktury datagramu

Zgodnie z zaleceniem zadaliśmy o to, by pakiety UDP były możliwie jak najkrótsze i przekazywały treści w zwięzły i jednoznaczny sposób. Poniżej zamieściliśmy poglądowy rysunek struktury datagramu z wiadomością zwrotną zawierającą wskazanie potencjometru:



Rysunek 4 Struktura wiadomości RESPONSE

Ponieważ wskazanie potencjometra przechowywane jest na 10 bitach, najbliższą pełną wartością bajtów jest 16. Oznacza to, że do wykorzystania pozostaje 6 bitów. Jak widać na powyższym obrazku postanowiliśmy podzielić pakiet na bloki:

- 10 bitów przeznaczonych na wartość zmierzonego napięcia na wejściu A2 (najmłodsze bity)
- 4 bity (idąc w kierunku najstarszego bitu) na identyfikator datagramu
- 2 bity na flagę (wiadomości HELLO, REQUEST, RESPONSE)

W ten sposób przesyłany datagram będzie miał 16 bitów, czyli 2 bajty. W naszym rozwiązaniu pierwszym bajtem jest „prawy” zawierający główną część wskazania potencjometru, a drugim kolejna część danych.

Aby zminimalizować przesyłane dane, wiadomości REQUEST i RESPONSE mają tylko jeden bajt. Ich podział wygląda następująco:

- 2 najstarsze bity: flaga
- 4 środkowe bity: numer pakietu
- 2 najmłodsze bity: niewykorzystane

Numeracja pakietu może osiągnąć maksymalną wartość 15, po tym jest resetowana do wartości 1. Wprowadziliśmy takie rozwiązanie, aby wolne 4 bity nie zostały zmarnowane oraz aby osiągnąć pewną synchronizację. W przypadku gdy numery nie będą się zgadzały zostanie o tym wyświetlona informacja na porcie szeregowym.

Opis działania

Opracowaliśmy kod klienta (dla emulatora Arduino) oraz serwera (pod Arduino), oba znajdują się w repozytorium.

Zgodnie z treścią zadania serwer ogłasza gotowość przesyłając datagram powitalny HELLO, który w naszym przypadku jest pakietem z ustawioną flagą: 0b10 na dwóch najstarszych bitach.

Następnie serwer przechodzi w stan oczekiwania na żądanie od klienta i gdy otrzyma pakiet z flagą REQUEST_MESSAGE 0b01, odsyła odpowiedź z flagą RESPONSE_MESSAGE 0b11 oraz w 10 bitowym bloku wartość zmierzonego przez potencjometr napięcia.

Wysłany raport odbierany jest przez klienta, który dalej może wysyłać następne żądania.

Ponadto analogicznie jak na zaprezentowanym wyżej obrazku przedstawiającym strukturę datagramu każdy pakiet zawiera 4-bitowy identyfikator, który go wyróżnia i pozwala na identyfikację oraz synchronizację.

Działanie

Na poniższym rysunku prezentujemy poprawne działanie naszego rozwiązania. Po uruchomieniu serwer wysyła wiadomość HELLO z numerem datagramu 1. Następnie klient odbiera ją, wysyła żądanie i oczekuje na odpowiedź. Następnie na zmianę klient i serwer wymieniają się wiadomościami REQUEST i RESPONSE. Można zaobserwować, że wartość wskazania potencjometru rzeczywiście zmienia swoją wartość i jest poprawnie przesyłana oraz odbierana. Klient odebrawszy każdy z raportów, demonstruje ich treść z wykorzystaniem swojej konsoli szeregowej, co umożliwia diagnozowanie czasów kiedy taki raport odebrano.

```

COM4
Connecting to WiFi.....Connected to WiFi!
IP Address : 192.168.0.50
+++Sent hello message to 192.168.0.51 +++
===Server has received data request
Packet num: 2
===
---Sending response
Read value: 596
Packet number: 3
---
===Server has received data request
Packet num: 4
===
---Sending response
Read value: 946

[ ] Autoscroll [ ] pokaż znacznik czasu Nowa linia 9600 baud Wyczyść okno

UART
192.168.0.51
+++Received hello message
Received packet number: 1
+++
---Sending request
Packet number: 2
---
===Client has received response. time: 15543
Packet number: 3
Potentiometer value: 596
===
---Sending request
Packet number: 4
---
===Client has received response. time: 18860
Packet number: 5
Potentiometer value: 946
===
  
```

Rysunek 5 Nawiązanie połączenia klienta i serwera

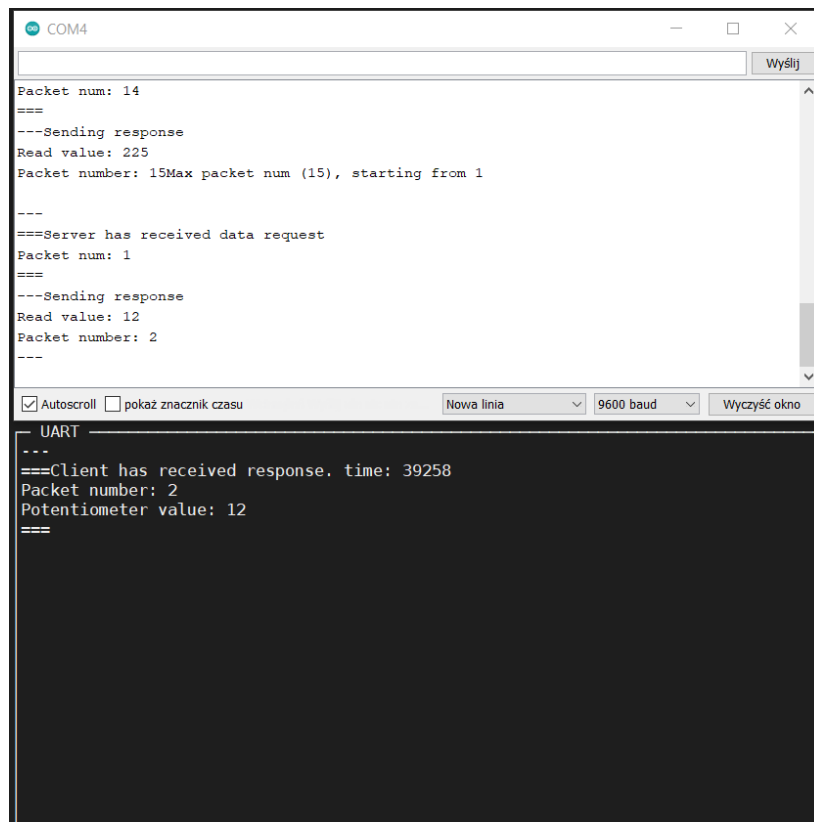
Czas pomiędzy każdym wysłanym żądaniem wynosi w przybliżeniu wymaganą wartość 3380 ms.

Przykładowy okres: 1707.465875 ms – 1704.087039 ms = 3.378836 ms ≈ 3380 ms.

315	1704.087039	192.168.0.50	192.168.0.51	UDP	60 8276 → 8276 Len=2
316	1707.396131	192.168.0.51	192.168.0.50	UDP	60 8276 → 8276 Len=1
317	1707.465875	192.168.0.50	192.168.0.51	UDP	60 8276 → 8276 Len=2

Rysunek 6 Czas pomiędzy żądaniami

W przypadku gdy numer wiadomości osiąga maksymalną wartość 15, zdarzenie takie jest wykrywane po obu stronach oraz wartość identyfikacyjna następnego pakietu ustawiana jest na 1. Synchronizacja jest poprawna, co można zaobserwować na poniższym zrzucie ekranu. Serwer zresetował wartość id, a klient od razu był przygotowany i wysłał wiadomość z id równym 1.



```
COM4
Packet num: 14
===
---Sending response
Read value: 225
Packet number: 15Max packet num (15), starting from 1
---
===Server has received data request
Packet num: 1
===
---Sending response
Read value: 12
Packet number: 2
---
UART
---
===Client has received response. time: 39258
Packet number: 2
Potentiometer value: 12
===
```

Rysunek 7 Poprawne przejście identyfikatorów

Podsumowanie

Dzięki zadaniom zaplanowanym w ramach 2 laboratorium z przedmiotu PSIR, w sposób praktyczny przyswoiliśmy wiedzę z wykładu dotyczącą tworzenia i uruchamiania oprogramowania sieciowego w systemie Linux bazującego na socketach. Udało nam się z sukcesem zaimplementować oraz przetestować rozwiązania zaplanowanych zadań, działające na emulatorze Arduino oraz na układzie ESP8266.