Data 15.01.2024

Tuple Space Middleware for Distributed Computing

Autorzy:

Bartłomiej Miłkowski

Jan Dziewulski

Filip Jaworski

Spis treści

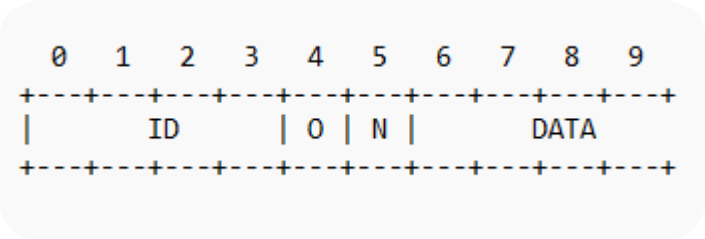
Specyfikacja ALP

ALP, czyli protokół warstwy aplikacji jest kluczowym elementem w komunikacji między klientem a serwerem, utrzymującym przestrzeń krotek. Jego głównym celem jest

umożliwienie komunikacji i koordynacji opartej na przestrzeni krotek. Jest on niezależny od języka i platformy, co oznacza, że klienci oparci na różnych platformach, napisani w różnych językach, mogą uczestniczyć w aplikacji rozproszonej wykorzystującej przestrzeń krotek. Umożliwienie różnym stronom uczestnictwa jest korzyścią z dobrze określonego protokołu i jednym z głównych powodów jego implementacji w systemie.

Wiadomości ALP mają następujący format:

| Typ operacji | ID krotki | Liczba pól | Typy pól | Dane pól |



O – rodzaj operacji (prośba o udostępnienie krotki/prośba o wstawienie krotki do przestrzeni krotek)

N – ilość pól danych krotki

Gdzie:

Typ operacji to jednobajtowy identyfikator operacji, który może przyjmować wartości `TS_OUT`, `TS_INP` lub `TS_RDP`.

ID krotki to czterobajtowy identyfikator krotki.

Liczba pól to jednobajtowy identyfikator określający liczbę pól w krotce.

Typy pól to jednobajtowe identyfikatory typów pól. Możliwe wartości to `TYPE_INT` i `TYPE_FLOAT`.

Dane pól przechowują wartości. Każde pole może mieć rozmiar czterech bajtów (dla `int (Dla systemu Linux int – 4 bajty, dla Arduino long – 4 bajty, co zostało zamienione z tego powodu)` i `float – 4 bajty`).

Pola te bezpośrednio odpowiadają atrybutom zawartym w kodzie, które zostały określone w następujący sposób:

- **Operation Type** – char operation type;

- **ID** – unsigned int id;

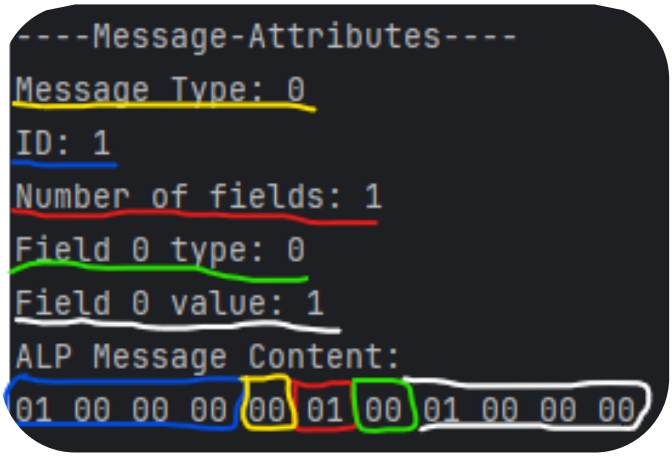
- **Num Fields** – char num_fields;

- **Data Type** – char data_type;

- **Value** – int/long lub float data;

Wiadomości w naszym protokole ALP są kodowane w formacie binarnym i składają się z kilku segmentów. Każdy segment reprezentuje różne części informacji o krotce. Oto jak zostało to określone w kodzie:

1. **Typ operacji** (1 bajt): Pierwszy bajt wiadomości reprezentuje typ operacji. Może to być `TS_OUT` (0), `TS_INP` (1) lub `TS_RDP` (2).
2. **ID krotki** (4 bajty): Następne cztery bajty reprezentują ID krotki jako liczbę całkowitą.
3. **Liczba pól** (1 bajt): Kolejny bajt reprezentuje liczbę pól w krotce.
4. **Typy pól** (1 bajt na pole): Kolejne bajty reprezentują typy pól. Każde pole ma swój typ reprezentowany jako `TYPE_INT` (0) lub `TYPE_FLOAT` (1).
5. **Dane pól** (4 bajty na pole): Kolejne segmenty reprezentują dane pól. Każde pole ma swoje dane reprezentowane jako liczba całkowita (dla `TYPE_INT`) lub liczba zmiennoprzecinkowa (dla `TYPE_FLOAT`).



Atrybuty składowe wiadomości protokołu ALP przedstawione za pomocą terminala.

Taki sposób przesyłania danych jest barzdiej optymalny i uniwersalny. Gdyby przenosić wszystkie dane w postaci struktury tuple_t, byłaby potrzeba zużycia dużo większej ilości

bajtów. W tym rozwiązaniu przyjmujemy na przykład, że na informację o typie operacji między klientem i serwerem (*Message Type*) oraz ilości pól jakie przesyłamy (*Number of fields*) przekazujemy po jednym bajcie. Dodatkowo nasza struktura tuple_t ma z góry zaalokowaną pamięć pozwalającą przechowywać do dziesięciu elementów, jednak w wielu przypadkach klient umieszcza w krotce dużo mniej. Gdyby nie stosować metody wysyłania wiadomości binarnie, przesłanoby w takiej sytuacji nawet do 36 pustych, niepotrzebnych bajtów.

Opis implementacji API tuple space (dla węzła IoT)

Interfejs API przestrzeni krotek (tuple_space.c / tuple_space.cpp) w systemie składa się z trzech funkcji: ts_out, *ts_inp* i *ts_rdp*. Te funkcje są używane do interakcji z przestrzenią krotek. Oto szczegółowy opis implementacji tych funkcji:

- *ts_out(int id, field_t* fields, int num_fields*): Ta funkcja jest używana do umieszczania krotek w przestrzeni krotek. Przyjmuje identyfikator krotki, tablicę pól i liczbę pól jako argumenty. Najpierw tworzy wiadomość do wysłania do serwera, zawierającą te informacje. Następnie wysyła tę wiadomość do serwera za pomocą funkcji send_message_to_server.
- *ts_inp(int id, field_t* fields, int num_fields)*: Ta funkcja jest używana do pobierania krotek z przestrzeni krotek. Przyjmuje identyfikator krotki, tablicę pól i liczbę pól jako argumenty. Podobnie jak ts_out, tworzy wiadomość do wysłania do serwera. Jednak oprócz tego oczekuje na odpowiedź od serwera. Kiedy otrzyma odpowiedź, dekoduje ją i aktualizuje pola na podstawie otrzymanych danych. Jest to funkcja blokująca, będzie nasłuchiwała tak długo dopóki nie dostanie zwrotnej krotki. Znaczy to, że klient, który wysłał prośbę o krotkę za pomocą tej funkcji, będzie nasłuchiwał co 5 sekund, wysyłając kolejną prośbę, dopóki nie zostanie zwrócona mu krotka.
- *ts_rdp(int id, field_t* fields, int num_fields):* Funkcja ta jest praktycznie identyczna do funkcji *ts_inp.* Jedyną różnicą jest flaga wysyłana w

treści wiadomości, która informuje o tym, że chce jedynie odczytać krotkę bez jej usuwania. Jest to również funkcja blokująca.

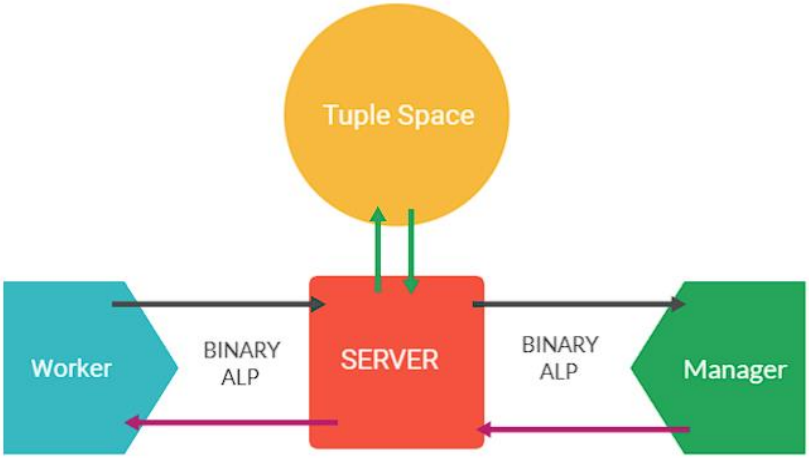
W kontekście IoT, te funkcje mogą być używane do komunikacji między różnymi urządzeniami IoT. Na przykład, urządzenie IoT może używać funkcji *ts_out* do umieszczenia danych sensora w przestrzeni krotek, a inne urządzenie IoT może używać funkcji *ts_inp* lub *ts_rdp* do odczytania tych danych. Dzięki temu urządzenia IoT mogą łatwo i efektywnie współdzielić dane. Co razem z implementacją protokołu ALP pozwala na stworzenie zespołu urządzeń, które mogą wymieniać się danymi, bez względu na język programowania zastosowany do stworzenia poszczególnych aplikacji. Ograniczeniem systemu jest natomiast typ wymienianych poprzez krotki danych. Ta implementacja zawiera definicje tylko dwóch typów danych takich jak *int* i *float* z pominięciem takich typów jak *string* czy *bolean*. Jest to zamierzone działanie, gdyż nie przewidziano połączenia middleware z aplikacją zewnętrzną, która potrzebowałaby takiej implementacji.

Na czym polega elastyczność rozwiązania?

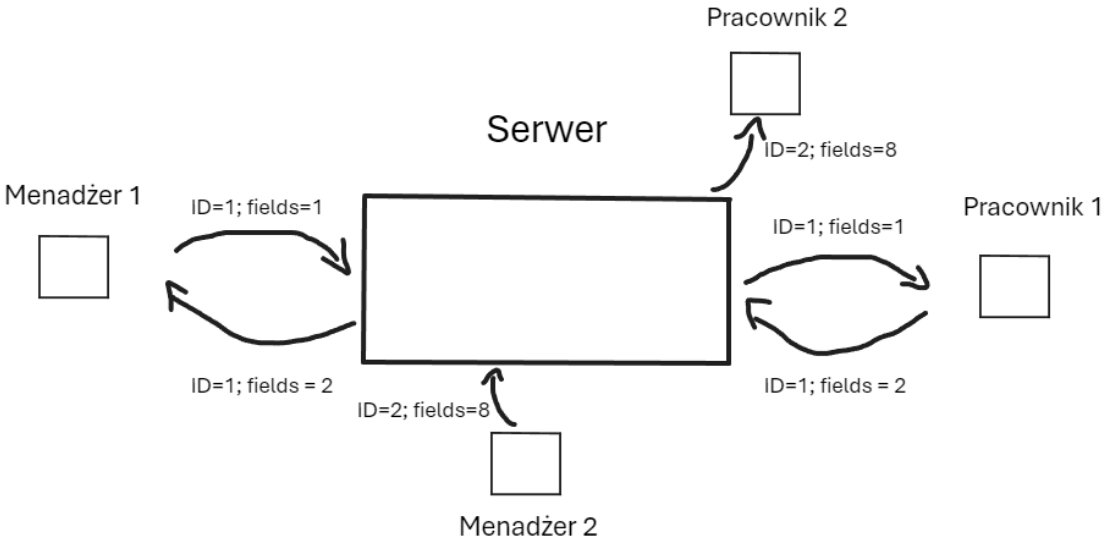
Początkowym pomysłem było umieszczenie w krotce dodatkowego pola *status,* które informowałoby czy zadanie wysłane przez klienta ma być odbierane i obsługiwane przez pracownika, czy jest już gotowe do odbioru przez managera. Byłby to jednak mechanizm nie tyle nieuniwersalny, ponieważ może dołączyć klient, który nie będzie chciał z powrotem odbierać krotek, a być może będzie chciał wymieniać informacje między managerem i workerem kilkukrotnie. Dodatkowo sprawia to, że w systemie IoT jesteśmy zmuszeni do przesyłania i przechowywania kolejnego bajtu danych na każdą jedną krotkę co w przypadku wielu krotek może być bardzo kosztowne. Zamiast tego pomysłu, zdecydowano, aby w przypadku prośby do serwera o przydzielenie krotki, krotka była dopasowywana pod względem nie tylko ID zadania, ale również była dopasowana do życzenia klienta, miała określoną ilość danych, oraz określone ich typy. Dzięki takiemu rozwiązaniu, wystarczy, że pracownik będzie dopisywał do krotki wynik zadania dodając jedno pole, a manager odbierał tylko te krotki z dodatkowym polem. Jeżeli zadanie

będzie wymagało jedynie odczytu wysłanych przez managera danych bez ich odsyłu, jest to tak samo możliwe.

Schemat rozwiązania



Schemat poglądowy



Opis implementacji serwera

Serwer został zaimplementowany w pliku `*server.c,*` składającym się z kilku kluczowych komponentów:

Gniazdo sieciowe: Serwer używa gniazda UDP do komunikacji z klientami. Gniazdo jest tworzone na początku działania serwera i jest używane do odbierania i wysyłania wiadomości.

Funkcje obsługi wiadomości: Serwer zawiera kilka funkcji do obsługi wiadomości, takich jak *printMessage*, *printTuple*, *convertTupleToMsg*, *sendNotFoundTupleMessage*, *sendTuple*, *deleteTuple*, *decode_message_from_client* i *addTuple*. Te funkcje są używane do interpretacji wiadomości od klientów, manipulacji krotkami w przestrzeni krotek i wysyłania odpowiedzi do klientów.

Pętla główna: Serwer działa w nieskończonej pętli, oczekując na wiadomości od klientów, dekodując te wiadomości, wykonując odpowiednie operacje na przestrzeni krotek i wysyłając odpowiedzi do klientów.

Główne struktury danych obsługiwanych przez serwer:

Przestrzeń krotek: Przestrzeń krotek jest reprezentowana jako tablica struktur *tuple_t*. Każda struktura *tuple_t* reprezentuje jedną krotkę i zawiera identyfikator krotki, liczbę pól, typy pól i dane pól. Tablica jest zaimplementowana statycznie. Trzeba zaalokować odpowiednią ilość pamięci na przechowywanie krotek. Na potrzeby testów ustalono długość tablicy 512. W celu rozwinięcia rozwiązania, należałoby napisać strukturę danych przypominającą listę, bądź kopiec, dynamicznie alokującą i zmniejszającą pamięć.

Nasza struktura pozwala na wyszukiwanie, odczytywanie, dodawanie oraz usuwanie krotek.

Dodawanie odbywa się poprzez dodanie zadania do kolejki, za ostatnim elementem w tablicy.

Usuwanie polega na zamianie indeksami elementu ostatniego, z tym, który ma zostać usunięty, następnie jest usuwany. Dzięki temu rozwiązaniu, zawsze usuwany jest ostatni element, a w tablicy nie występują „dziury” sprawiające szybsze zapełnianie się pamięci i brak kontroli nad nią.

Wszystkie funkcje obsługujące tablicę przechowującą krotki zostały opisane poniżej.

Wiadomości ALP: Wiadomości są reprezentowane jako tablice bajtów. Każda wiadomość zawiera typ operacji, identyfikator krotki, liczbę pól, typy pól i dane pól.

Szczegółowy opis funkcji serwera:

- ***printMessage***: Ta funkcja wyświetla wiadomość w formacie binarnym. Jest używana do debugowania i sprawdzania poprawności wiadomości.

- ***printTuple***: Ta funkcja wyświetla wybraną po indeksie krotkę. Jest używana do debugowania i sprawdzania poprawności krotek.

- ***isTupleSpaceEmpty***: Ta funkcja sprawdza, czy przestrzeń krotek jest pusta. Jest używana do kontrolowania stanu przestrzeni krotek.

- ***convertTupleToMsg***: Ta funkcja konwertuje krotkę na wiadomość, która może być wysłana przez sieć. Jest używana do przygotowania krotek do wysłania.

- ***sendNotFoundTupleMessage***: Ta funkcja wysyła wiadomość do klienta, informując go, że żądana krotka nie została znaleziona. Jest używana, gdy operacja `*inp*` lub `*rdp*` nie może znaleźć pasującej krotki.

- ***sendTuple***: Ta funkcja wysyła krotkę do klienta. Jest używana, gdy operacja `*inp*` lub `*rdp*` znajduje pasującą krotkę.

- ***deleteTuple***: Ta funkcja usuwa krotkę z przestrzeni krotek. Jest używana po operacji `*inp*`.

- ***decode_message_from_client***: Ta funkcja dekoduje wiadomość od klienta. Jest używana do interpretacji żądań od klientów.

- ***addTuple***: Ta funkcja dodaje krotkę do przestrzeni krotek. Jest używana po operacji `*out*`.

Wszystkie te funkcje współpracują ze sobą, aby umożliwić serwerowi obsługę operacji na przestrzeni krotek. Serwer oczekuje na żądania od klientów, dekoduje te żądania, wykonuje odpowiednie operacje na przestrzeni krotek, a następnie wysyła odpowiedzi do klientów.

Opis funkcjonalności i implementacji aplikacji App1

Pierwsza aplikacja App1, odpowiadająca pierwszemu węzłowi IoT, składa się z dwóch procesów: Manager, który wysyła dwie dane typu *int* i Worker'a, który dzieli jedną przez drugą. Następnie Worker dopisuje do otrzymanej od Manager'a krotki wynik jako zmienna typu *float* i odsyła ją do przestrzeni krotek. Na końcu Manager odczytuje odesłaną krotkę, poznając wynik pracy procesu Worker'a. Ta aplikacja ma na celu zilustrowanie jednego ze sposobów implementacji struktury rozproszonej master/worker, ktorą w naszym przypadku nazwaliśmy manager/worker.

Program Manager:

- Tworzy dwa pola typu int o wartościach podanych przez nas wartościach testowych 38 i 20.
- Wysyła te pola do przestrzeni krotek za pomocą funkcji ts_out.
- Czeka 5 sekund.(Jest to czas określony przez nas na potrzeby testu)
- Próbuje odczytać z przestrzeni krotek krotkę składającą się z dwóch pól typu int i jednego pola typu float za pomocą funkcji ts_rdp. W przypadku niepowidzenia, ponawia proces.
- Wyświetla wynik odczytany z pola typu float.

Program Worker:

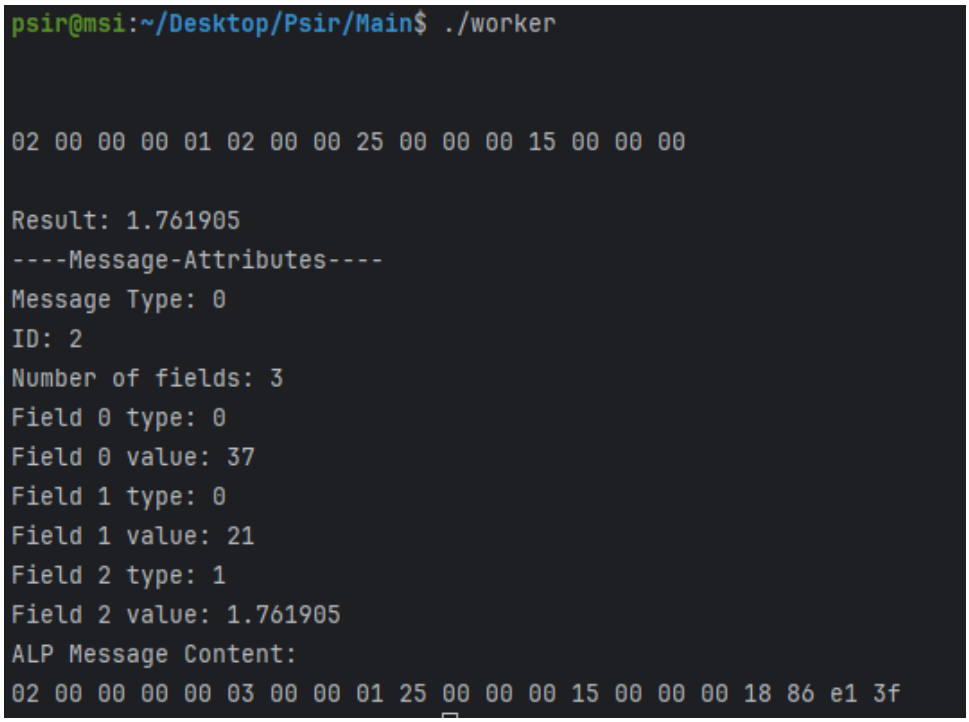
- Próbuje odczytać, nasłuchując, z przestrzeni krotek krotkę składającą się z dwóch pól typu int za pomocą funkcji ts_inp.
- Oblicza wynik dzielenia wartości pierwszego pola przez wartość drugiego pola i zapisuje go jako float.
- Wyświetla wynik.
- Odsyła do przestrzeni krotek krotkę składającą się z dwóch pól typu int (te same, które wcześniej odczytał) oraz jednego pola typu float (wynik obliczeń) za pomocą funkcji ts_out.

Poniżej znajdują się wyniki przeprowadzonych przez nas testów dla aplikacji APP1

```
psir@msi:~/Desktop/Psir/Main$ ./manager

----Message-Attributes----
Message Type: 0
ID: 2
Number of fields: 2
Field 0 type: 0
Field 0 value: 37
Field 1 type: 0
Field 1 value: 21
ALP Message Content:
02 00 00 00 00 02 00 00 25 00 00 00 15 00 00 00

02 00 00 00 01 03 00 00 01 25 00 00 00 15 00 00 00 18 86 e1 3f

Result: 1.761905
```

Obserwujemy zachowanie systemu z perspektywy procesu Manager'a.

W wyniku przeprowadzonego testu widzimy, że wiadomość wysyłana jest w postaci ciągu binarnego, który reprezentuje krotkę, która została utworzona w przestrzeni krotek. Po upływie 5s odczytujemy wiadomość zwrotną z dodanym wynikiem działania procesu Worker'a.



Obserwujemy zachowanie systemu z perspektywy procesu Worker'a.

W wyniku przeprowadzonego testu widzimy, że wiadomość wysyłana jest w postaci ciągu binarnego, który reprezentuje krotkę, która została przesłana w odpowiedzi do utworzonej przez Managera krotki. Jesteśmy w stanie odczytać jaka wartość została uzyskana po podzieleniu dostarczonych przez Managera wartości, jak również obserwujemy ciąg binarny poszerzony o zapis dodatkowego pola z wynikiem działania programu.

Opis funkcjonalności i implementacji aplikacji App2 (GPIO - Arduino)

Aplikacja App2, czyli drugi węzeł IoT: Manager zapisuje do krotki w floatach temperaturę i wilgotność. Proces Workera odbiera (usuwając krotkę) i wypisuje otrzymane wyniki.

Ta aplikacja składa się z dwóch różnych programów, które komunikują się ze sobą za pomocą mechanizmu krotek. Oto, co robią poszczególne programy:

Program Manager2:

- Tworzy dwa pola typu `float` o wartościach 18.75 i 0.32.
- Wysyła te pola do przestrzeni krotek za pomocą funkcji `ts_out`.

- Czeka 1 sekundę.

Program Worker2:

- Próbuje odczytać z przestrzeni krotek krotkę składającą się z dwóch pól typu `float` za pomocą funkcji `ts_inp`.
- Wyświetla wartości odczytane z obu pól jako temperaturę i wilgotność.
- Czeka 3 sekundy.

Wszystkie programy komunikują się ze sobą za pomocą przestrzeni krotek, która działa na porcie 5012 na lokalnym hoście. Każdy program korzysta z tego samego identyfikatora krotek (`ID`), co oznacza, że mogą odczytywać i zapisywać te same krotki. Wszystkie programy korzystają z biblioteki `tuple.h`, która zawiera definicje typów i funkcji używanych do komunikacji za pomocą krotek.

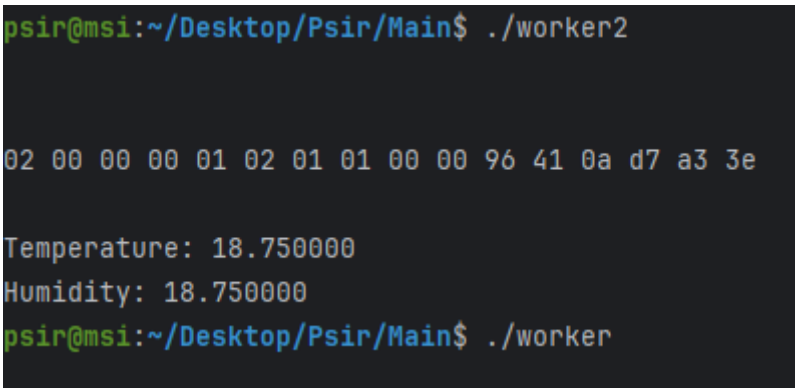
Aplikacja Linux

Przed implementacją programu App2 na węźle ze środowiskiem Arduino, przeprowadzone zostały testy na przykładowych danych, które zostały zawarte w kodzie aplikacji Linux (podobnie jak miało to miejsce przy App1).

```
psir@msi:~/Desktop/Psir/Main$ ./manager2

----Message-Attributes----
Message Type: 0
ID: 2
Number of fields: 2
Field 0 type: 1
Field 0 value: 18.750000
Field 1 type: 1
Field 1 value: 0.320000
ALP Message Content:
02 00 00 00 00 02 01 01 00 00 96 41 0a d7 a3 3e
```

Obserwujemy zachowanie systemu z perspektywy procesu Manager2'a.

W wyniku przeprowadzonego testu widzimy, że wiadomość wysyłana jest w postaci ciągu binarnego, który reprezentuje krotkę, która zawiera utworzone przez nas dane o temperaturze i wilgotności powietrza.



Obserwujemy zachowanie systemu z perspektywy procesu Worker'a.

W wyniku przeprowadzonego testu widzimy, że wiadomość wysyłana jest w postaci ciągu binarnego, który reprezentuje krotkę, która została przesłana w odpowiedzi do utworzonej przez Managera krotki. Jesteśmy w stanie odczytać jakie wartości zostały odczytane np. Sensor na urządzeniu takim jak Arduino.

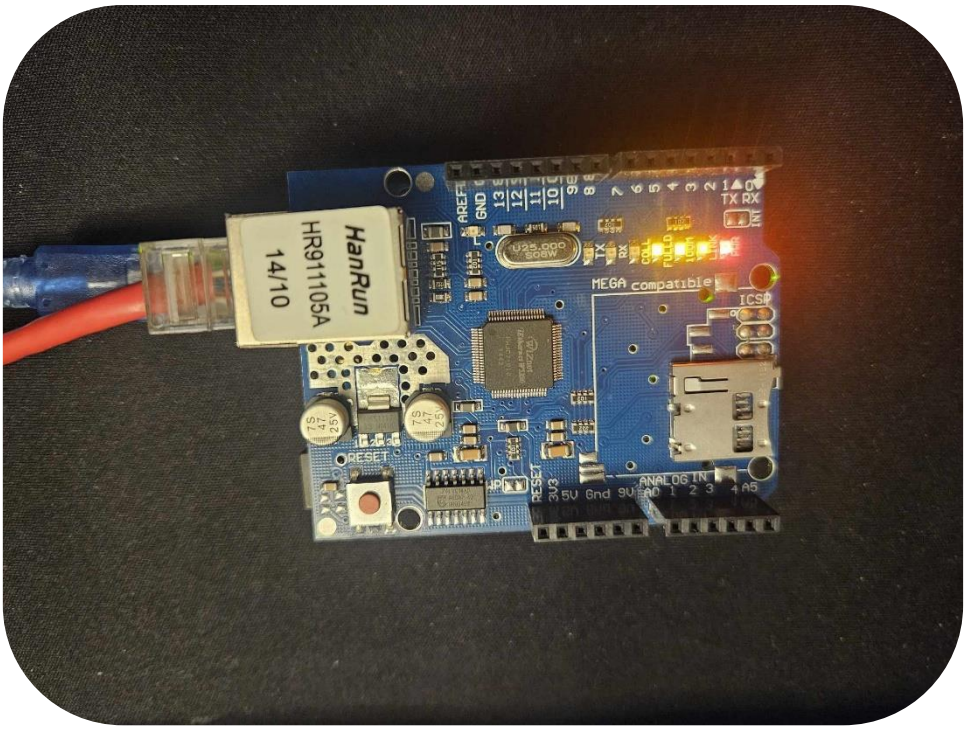
Aplikacja Arduino

Nasza implementacja aplikacji na Arduino składa się z dwóch głównych części: `setup()` i `loop()`. Oto, co robią te części:

Funkcja *`setup()`*:
- Inicjalizuje generator liczb losowych za pomocą wartości odczytanej z analogowego pinu 0.

Funkcja *`loop()`*:
- Tworzy tablicę *`templateArray`* składającą się z jednego pola typu `int`.
- Wyświetla typ danych i wartość odczytaną z *`templateArray`*`[0]`.
- Tworzy tablicę *`fields`* składającą się z dwóch pól typu `int` o wartościach 16 i 17(wartości testowe).
- Czeka 3 sekundy.
- Wysyła tablicę *`fields`* do przestrzeni krotek za pomocą funkcji *`ts_out`*.

Aplikacja ta korzysta z bibliotek Ethernet.h, EthernetUdp.h, tuple.h i tuple_space.h, które zawierają definicje typów i funkcji używanych do komunikacji sieciowej i zarządzania przestrzenią krotek.



Fizyczne Arduino, na którym przeprowadzono pierwsze testy

Opis plików środowiskowych dla aplikacji App2(GPIO - Arduino)

Pliki środowiskowe, znane również jako zmienne środowiskowe, to ciągi zawierające informacje o środowisku systemu oraz o aktualnie zalogowanym użytkowniku. Niektóre programy korzystają z tych informacji w celu stwierdzenia, gdzie umieszczać pliki (takie jak pliki tymczasowe). Zmienne środowiskowe mogą wpływać na działanie procesów uruchamianych w systemie operacyjnym i stając się pewnym mechanizmem komunikacji lub też przechowywać wartość w celu jej późniejszego wykorzystania.

W kontekście naszej aplikacji Arduino, pliki środowiskowe zawierają informacje takie jak:

- Adres IP i port serwera, z którym aplikacja powinna się komunikować.
- Identyfikatory używane do oznaczania krotek w przestrzeni krotek.
- Inne stałe lub zmienne, które są używane w całym kodzie.

Pliki te są załadowane na początku wykonania programu i dostarczają informacji konfiguracyjnych, które są używane przez resztę kodu. Niektóre z tych informacji są

zdefiniowane bezpośrednio w kodzie (np. `ID` i adres IP serwera), ale mogłyby być przeniesione do plików środowiskowych dla większej elastyczności i łatwiejszego zarządzania konfiguracją.

Środowisko działania aplikacji na fizycznym Arduino

Wszystkie części programu mogą ze sobą współpracować dzięki podłączeniu do wspólnej sieci. Serwer przestrzeni krotek został uruchomiony na maszynie wpiętej do tej samej sieci co fizyczne Arduino.

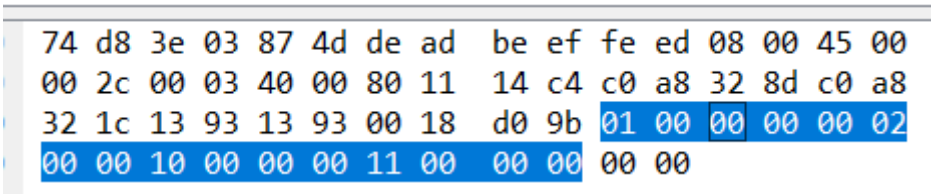
Test komunikacji funkcji ts_out

W celu przetestowania funkcji ts_out, został wybrany adres komputera z uruchomionym programem Wireshark. Wiadomość wysłana, która została odczytana poprzez serial monitor prezentuje się następująco:

```
02:50:44.800 -> ----Message-Attributes----
02:50:44.847 -> Message Type: 0
02:50:44.847 -> ID: 1
02:50:44.847 -> Number of fields: 2
02:50:44.894 -> Field 0 type: 0
02:50:44.894 -> Field 0 value: 16
02:50:44.941 -> Field 1 type: 0
02:50:44.941 -> Field 1 value: 17
02:50:44.941 -> ALP Message Content:
02:50:44.987 -> 1 0 0 0 0 2 0 0 10 0 0 0 11 0 0 0
```

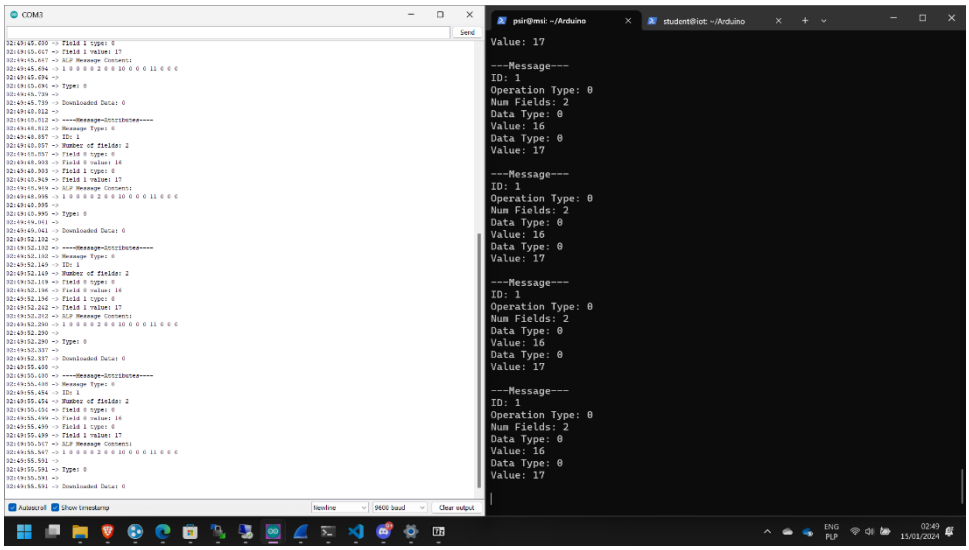
Treść wysłanej wiadomości

Wiadomość wysłana: 01 00 00 00 00 02 00 00 10 00 00 00 11 00 00 00.



Treść odebranej wiadomości w programie Wireshark

Odebrana wiadomość jest poprawna. Weryfikacja działania, pozwoliła przetestować działanie tej funkcji w komunikacji z serwerem.



Test implementacji ts_out na fizycznym Arduino wysłane dwie liczby typu całkowitego: 16 i 17

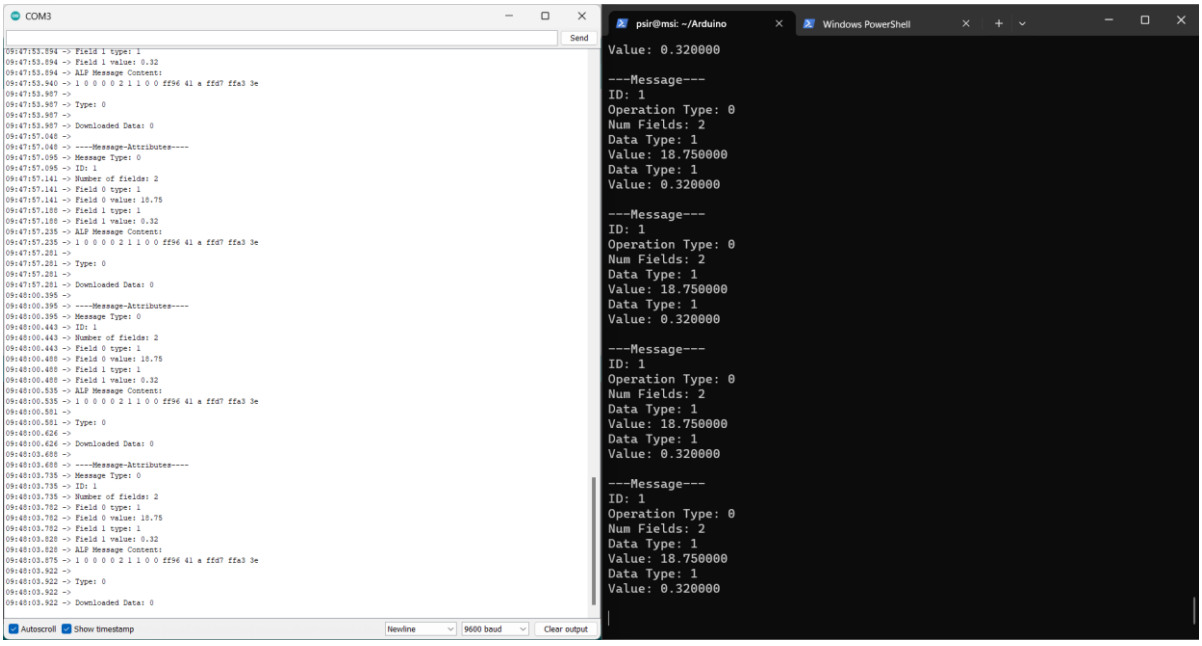
Można zauważyć, że wysłane jak i odebrane dane są poprawne.

Przesłanie temperatury i wilgotności z Arduino

Na początku zostały wysłane dwie liczby zmiennoprzecinkowe.

```
fields[0].dataType = TYPE_FLOAT;
fields[0].data.float_type = 18.75;
fields[1].dataType = TYPE_FLOAT;
fields[1].data.float_type = 0.32;
```

Liczby zostały przypisane w następujący sposób.



Zrzut potwierdzający przesyłanie danych z fizycznego Arduino na aplikację serwerową.

Arduino – ts_inp i ts_rdp

Niestety wystąpił pewien problem z odczytem danych z bufora udp, co uniemożliwiło uruchomienie tych funkcji (ts_inp, ts_rdp) na Arduino. Został znaleziony błąd podczas odczytu danych z bufora. W funkcji receive_message_from_server w linii Udp.read(msg, len).

```
int receive_message_from_server(char *msg)
{
    int len = Udp.parsePacket();
    if (len > 0)
    {
        Udp.read(msg, len);
        msg[len] = '\0';
        Serial.println("Received message from server:");
        Serial.println(msg);
    }

    return len;
}
```

Zrzut problematycznej funkcji

Udało się przeprowadzić na Arduino komunikację w jednym kierunku.