PSIR - Projekt

Przestrzeń krotek

Prowadzący: Jarosław Domaszkiewicz

Autorzy:

Julian Uziembło
Konrad Majewski
Alberto Szpejewski

15 stycznia 2024

Spis treści

1 Wstęp

Tematem projektu była przestrzeń krotek - asocjatywna, logicznie dzielona pamięć, przestrzeń do współpracy i wymiany informacji pomiędzy programami korzystającymi z niej.
Naszym zadaniem było napisanie implementacji takiej przestrzeni jako serwera, z którym komunikują się klienty, które mogą wydawać mu polecenia wpisania i wyjęcia/odczytu poszczególnych krotek.

2 Użyte narzędzia

- Oracle VirtualBox
- Visual Studio Code
- MobaX Term
- Linux Ubuntu 22.04.3
- WSL - Windows Subsystem for Linux
- GCC - kompilator języka C
- EBSim - software'owy emulator Arduino
- Overleaf LaTeX Editor

3 Założenia podstawowe

Podstawą projektu miał być protokół warstwy aplikacji (*Application Layer Protocol*, ALP) potrzebny do przekazywania danych pomiędzy urządzeniami. Docelowo te urządzenia to miałyby być urządzeniami IoT, protokół ten zatem powinien być binarny, aby zajmować jak najmniej miejsca.

3.1 Krotki

Krotka to uporządkowany zbiór danych różnych typów. Na nasze potrzeby, pierwsze pole krotki to zawsze pole "name" typu string (ciąg znaków), którego nie wliczamy do ilości pól krotki (skoro zawsze ono musi wystąpić). Później występuje sekwencja pól typów INT lub FLOAT, które mogą być wypełniona ("pełnoprawna" krotka), albo nie (szablon krotki). To właśnie ta sekwencja pól, oprócz nazwy, identyfikuje krotkę w przestrzeni.
Szablony krotek potrzebne są przy operacjach wyjmowania krotek - służą za wzór dla programu, który dopasowuje do nich krotki z przestrzeni. Dwie krotki są dopasowane, jeśli:

1. Mają tą samą wielkość
2. Mają tą samą nazwę
3. i-te pole krotki 1 ma ten sam typ, co i-te pole krotki 2.
4. Co najmniej jedno z dwóch pól

Na potrzeby uproszczenia, wybraną krotkę można zapisać w postaci: `("nazwa", [typ1] [dane1/?], [typ2] [dane2/?])` itd. Na przykład:

```
("dodaj_2", int 34)
("stan_D3_D8", int 0, float 3.14)
```

Aby przedstawić szablon krotki, można posłużyć się znakiem zapytania ("?") w miejscu niektórych danych, np:

```
("dodaj_2", int ?)
("stan_D3_D8", int ?, float 9.21)
```

3.2 Podstawowe zapytania

1. `out` - "włóż" wybraną krotkę do przestrzeni krotek.
2. `in` - "wyjmij" krotkę pasującą do podanego szablonu z przestrzeni krotek. Zapytanie blokuje, dopóki pasująca krotka nie znajdzie się w przestrzeni.
3. `rd` - "odczytaj" krotkę pasującą do podanego szablonu z przestrzeni krotek. Zapytanie blokuje, dopóki pasująca krotka nie znajdzie się w przestrzeni.
4. `inp` oraz `rdp` - nieblokujące wersje poleceń `in` oraz `rdp`. Zwracają wartość logiczną odpowiadającą temu, czy krotka została odczytana, czy nie.

3.3 Specyfikacja ALP

Protokół warstwy aplikacji miał definiować konstrukcję pakietu, w którym przesyłane będą dane o krotkach, a także zbiór reguł, jakimi kierować się będą serwer oraz klienty przy przesyłaniu tych wiadomości. Pakiety te docelowo mają być przesyłane w pakietach protokołu UDP, który nie gwarantuje niezawodnego dojścia pakietu, dlatego zdefiniowane zostały dodatkowe mechanizmy kontroli poprawnego ich przesyłania.
W naszym przypadku, ALP został podzielony na pewne podstawowe części: pakiet składał się z krotki, która składała się z pól.
Schemat krotki wraz z jej polami:

Nazwa pola	Rozmiar pola	Dozwolone wartości	Opis
Czy zajęte (*occupied*)	1 bit	0 lub 1	Wartość logiczna: 1 - TAK, 0 - NIE
Typ pola (*tuple_type*)	3 bity	INT(0b001), FLOAT(0b010)	Typ danych przechowywanych w polu krotki.
Dane (*data*)	4 bajty	Liczba typu INT lub FLOAT	Przechowuje dane krotki w postaci binarnej, które można odczytać albo jako typ INT albo FLOAT (w zależności od pola *tuple_type*)

Nazwa pola	Rozmiar pola	Dozwolone wartości	Opis
Nazwa krotki (*name*)	32 bajty	Do 31 dowolnych znaków ASCII + '\0' (*null terminator*)	Nazwa krotki, służąca do jej identyfikacji w przestrzeni krotek.
Wielkość (*size*)	4 bajty	Liczba z przedziału: 0...2^32-1 (0...4294967295)	Ilość pól krotki (wyłączając pole nazwy)
Pola krotki (*fields*)	4*8 bajtów	Pole krotki (opisane powyżej)	Pole krotki, przechowujące dowolną wartość typu INT lub FLOAT

Rysunek 1: Tabela przedstawiająca schemat pojedynczej krotki (u dołu) oraz pojedynczego pola krotki (u góry).

Poniżej opisane są pola krotki, które wymagają dodatkowych wyjaśnień:

- Wskaźnik zajętości pola - krotki mogą być szablonami, do których system ma porównywać przechowywane krotki w celu wyjęcia tej właściwej. Dlatego przydatny jest znacznik, czy dane pole jest zajęte, czy powinno być odbierane jako "dowolne pole danego typu".
- Typ pola - jedna z dwóch wartości: INT (liczba całkowita 32-bitowa) lub FLOAT (liczba zmiennoprzecinkowa 32-bitowa). Na tą konstrukcję zostały przeznaczone 3 bity z 2 powodów: aby późniejsze ulepszenia były prostsze w implementacji, oraz aby układ bitów 0b000 oznaczał nieprawidłowe ustawienie typu - łatwiej zauważyć błąd.
- Dane - dane w jednym z wyżej opisanych typów danych. W implementacji przechowywane pod tym samym adresem w pamięci, żeby oszczędzać miejsce.
- Wielkość krotki - wskazuje, ile segmentów (poza segmentem z nazwę) posiadać będzie krotka. Taka informacja jest przydatna m.in. przy odczycie danych z krotki, a także przesyłaniu jej.

Na podstawie powyższej struktury danych składane są pakiety protokołu, za pomocą których dane mogą być przesyłane pomiędzy urządzeniami:

Nazwa pola	Rozmiar pola	Dozwolone wartości	Opis
Typ zapytania (*req_type*)	3 bity	empty(0b000), out(0b001), in(0b010), inp(0b011), rd(0b100), rdp(0b101)	Typ polecenia wydawanego przestrzeni krotek.
Flagi (*flags*)	5 bitów	ACK(0b00001), RETRANSMIT(0b00010), KEEPALIVE(0b00100), HELLO(0b01000), ERROR(0b10000)	Flagi dołączane do danego zapytania, specyfikujące wybrane akcje.
Numer pakietu (*num*)	24 bity	Liczby z przedziału: 0...2^24-1 (0...16 777 215)	Losowo generowany numer pakietu. Przydatny np. Przy odsyłaniu odpowiedzi na zapytania.
Krotka (*tuple*)	Wielkość krotki *(tutaj: 68 bajtów)*	Krotka (opisana powyżej)	Krotka: przechowuje przesyłane dane.

Rysunek 2: Konstrukcja pakietu protokołu przestrzeni krotek.

Poniżej opisane są niektóre z pól pakietu protokołu przestrzeni krotek, które wymagają dodatkowych wyjaśnień:

- Typ zapytania - odpowiada każdemu podstawowemu zapytaniu, jakie implementować ma protokół (`out`, `in`, `inp`, `rd`, `rdp`) lub zapytaniu "empty" - pustemu zapytaniu, którego cel uściślają flagi.
- Flagi - mogą mieć wiele znaczeń w zależności od kontekstu:
  - `ACK`: wystawiana przy każdej poprawnej odpowiedzi serwera, powinna też być wystawiona podczas odsyłania potwierdzenia dotarcia pakietu przez klienta.
  - `RETRANSMIT`: flaga wystawiana przez serwer podczas retransmisji pakietu.
  - `KEEPALIVE`: flaga dla pakietu wysyłanego pomiędzy wątkami serwera, które mają do serwera jakieś zapytanie (np. o dodanie pakietu do przestrzeni), wystawiana, aby serwer nie wydawał jeszcze polecenia zamknięcia wątku (więcej o tym mechanizmie w sekcji o działaniu serwera).
  - `HELLO`: flaga wystawiana przez klienta, który chce sprawdzić, czy serwer działa poprawnie.
  - `ERROR`: flaga wystawiana przez serwer w odpowiedzi do klienta w przypadku różnego rodzaju błędów, LUB, w szczególnym przypadku zapytań `inp` oraz `rdp`: informacja o nieznalezieniu danej krotki w przestrzeni (jeśli sparowana z flagą ACK).
- Numer pakietu: 24-bitowa liczba całkowita, przydatna przy odsyłaniu odpowiedzi na zapytania (odsyłana jest liczba o 1 większa (modulo 2^{24}) dla zapytań poprawnych).

Aby komunikacja przebiegała niezachwianie, stosujemy następujące reguły:

1. Dozwolone zapytania od klientów: protokół pozwala na poniższe zapytania od klientów (na inne zapytania serwer po prostu odeśle pustą odpowiedź z flagą ERROR, tak, jak jest to opisane niżej):

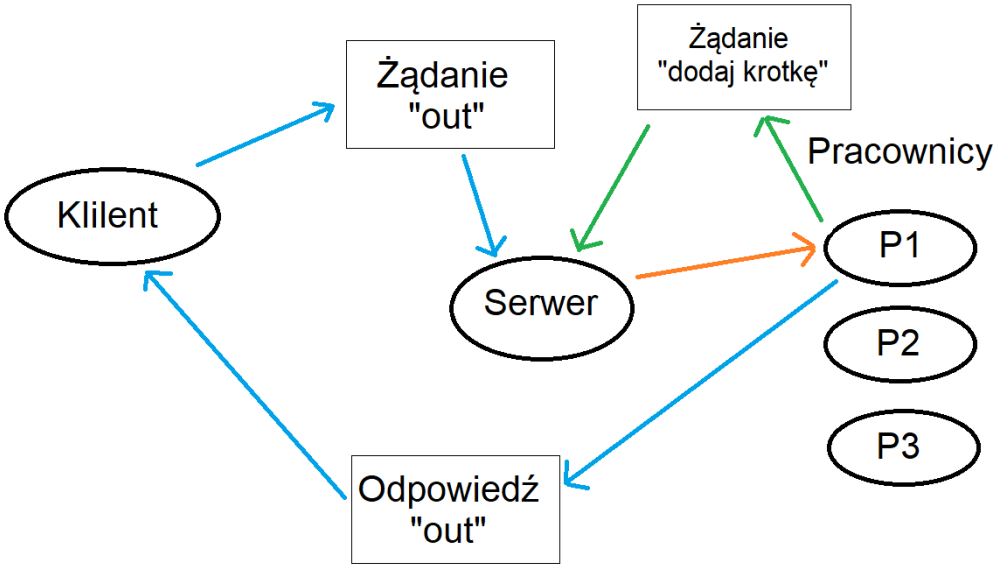
- zapytanie typu "empty" z flagą HELLO: "przywitanie" z serwerem, sprawdzenie, czy działa.
- zapytania typu "out", "in", "inp", "rd", "rdp" bez żadnych flag: poprawne zapytanie do serwera. Dodatkowo: zapytanie "out" musi zawierać krotkę, której każde pole jest wypełnione (inaczej: krotka nie może mieć pola z sekcją "occupied" oznaczoną jako "NO" (0)). Reszta zapytań może zawierać krotki w dowolnej postaci.
- zapytanie powyższych typów z flagą ACK i pustą krotką: potwierdzenie przyjęcia pakietu przez klienta. Wymagane wysłanie po każdym poprawnie odebranym pakiecie zwrotnym.

2. Odpowiedzi serwera:

- odpowiedź "empty" z flagą ERROR + flagami klienta: odpowiedź na błędne zapytanie od klienta.
- odpowiedź każdego innego typu:
  - z flagą ACK: wysyłane z każdą odpowiedzią na poprawne zapytanie, jako dowód przyjęcia zapytania.
  - z flagą ERROR: wysyłane, jeśli flagi zapytania są źle ustawione, albo jeśli wystąpił jakiś wewnętrzny błąd serwera.
  - z flagami ACK i ERROR (jednocześnie): wysyłane jako odpowiedź na zapytanie "inp" lub "rdp", jeśli zapytanie było poprawne, ale podana krotka nie została znaleziona.
  - z flagami ACK i HELLO: wysyłane jako odpowiedź na zapytanie HELLO.
  - z flagą RETRANSMIT: jeśli klient nie potwierdził "dojścia" pakietu, jest on retransmitowany z tą flagą, aby klient wiedział, że to retransmisja.
- *dodatkowo*: odpowiedź na każde poprawne zapytanie do serwera jest odsyłana z numerem pakietu zinkrementowanym o 1 (modulo 2^{24}), a każda odpowiedź na zapytanie błędne - z tym samym numerem, który dostała.

4 Działanie serwera

Nasza implementacja serwera działa na zasadzie skończonego zbioru tzw. "pracowników": procesów, które mogą być w 2 stanach: "idle" (oczekujący na żądanie) oraz "active" (wykonujących żądanie). Zbiór ten jest skończony, aby nie przeciążyć zasobów serwera - nie ma możliwości włączenia więcej, niż ustalona liczba procesów, naraz. Model ten przydaje się także przy poleceniach `in` oraz `rd`, które z zamierzenia mają blokować wątek. Aby więc serwer nadal odpowiadał na zapytania, potrzebne jest ustalenie takiego modelu pracy.
Implementacja tego modelu działa następująco: serwer przyjmuje żądanie od klienta i od razu oddelegowuje wątek do jego rozpatrzenia. Wątek parsuje żądanie, wykonując odpowiednie czynności. Aby wykonać operacje na przestrzeni krotek, "pracownik" wysyła uproszczone zapytanie do serwera (drogą komunikacji wewnętrznej - za pomocą Linux'owego pipe(2) API). Serwer wykonuje proste polecenie - wpisz krotkę do przestrzeni, odczytaj/wyjmij krotkę, i odsyła status operacji pracownikowi. Pracownik odsyła odpowiednią odpowiedź do klienta.



Rysunek 3: Uproszczony, poglądowy schemat działania serwera na przykładzie zapytania "out".

Serwer przechowuje krotki w postaci "stosu" - połączonej listy jednokierunkowej, w której każdą z krotek reprezentuje oddzielny węzeł listy. Krotki zawsze "wkładane" są na czubek listy. Aby wyjąć/odczytać krotkę z listy, trzeba przejrzeć wszystkie krotki w liście. Zdefiniowane operacje:

- `new`: tworzy nową, pustą instancję listy.
- `free`: zwalnia pamięć zalokowaną dla danej listy.
- `add`: dodaje krotkę do listy.
- `find`: znajduje krotkę odpowiadającą podanemu szablonowi i zwraca jej kopię.
- `withdraw`: znajduje krotkę odpowiadającą podanemu szablonowi i zwraca ją, usuwając ją z listy.

Taka struktura danych została wybrana głównie z powodu łatwości implementacji, ale też z powodu, że jest ona strukturą dynamiczną - dowolny węzeł listy może zostać w każdej chwili odczytany/wyjęty z listy, a nowe węzły dokładane mogą być w prosty sposób w czasie pracy programu.

5 Węzły IoT

Platformą węzłów IoT miało być Arduino. Pozwala ono w prosty sposób implementować podstawowe funkcjonalności węzłów. W tym wypadku główną funkcjonalnością, implementowaną przez wszystkie węzły, jest możliwość komunikacji z serwerem krotek. W każdym z węzłów zdefiniowane są potrzebne mu funkcje: out, in, inp, rd, rdp. Komunikują się one w ustandaryzowany sposób: wysyłają opisane wyżej pakiety z żądaniami, otrzymują na nie odpowiedź i odsyłają potwierdzenie (tak, jak to zostało opisane w sekcji 3.2 "Specyfikacja ALP").

6 Aplikacja 1 - model `master` – `worker`

Aplikacja 1 ma obrazować zasadę działania modelu `master` – `worker` przy użyciu przestrzeni krotek jako środka komunikacji. Proces `master` ma wrzucać do przestrzeni zadania (w tym przypadku - sprawdzenia, czy dana liczba jest liczbą pierwszą), a później zbierać ich wyniki. Proces `worker` ma za zadanie pobierać dostępne zadania, wykonywać je, i wrzucać ich wyniki do przestrzeni.
Dokładne opisy tych 2 procesów:

1. Proces `master`: wpisuje do przestrzeni krotek (za pomocą funkcji "out") krotki z nazwą "check_is_prime" oraz liczbami - od 2 do N (w tym przypadku: przyjęliśmy 10, ale jest to liczba którą można zmienić na dowolną). Po wpisaniu wszystkich krotek, zaczyna próbować wyciągać krotki (za pomocą funkcji "in" - proces ma blokować, aż nie wyciągnie danej liczby krotek): `("is_prime", int ?)` oraz `("is_not_prime", int ?)`, aż do wyciągnięcia N-1 takich krotek z przestrzeni. Na koniec wyświetla otrzymane wyniki.

2. Procesy `workers`: wyciąga z przestrzeni krotek (za pomocą funkcji "in" - blokuje wątek, aż nie wyciągnie) krotki o sygnaturze ("check_is_prime", int ?), sprawdza, czy dana liczba jest liczbą pierwszą, po czym wysyła do przestrzeni krotkę ("is_prime", int n), jeśli liczba n jest pierwsza, lub ("is_not_prime", int n), jeśli liczba nie jest liczbą pierwszą.

Działanie tej aplikacji zostało przedstawione w filmiku `app1_-_master-worker.mkv`, który znajduje się w repozytorium wraz z resztą projektu.

7 Aplikacja 2 - sensorsy i licznik

Aplikacja 2 ma obrazować zasadę działania systemu z wieloma sensorami i jednym licznikiem. Sensory mają za zadanie raportować o zmianie pinu GPIO (skonfigurowanego jako input) z 0 na 1 i na odwrót, i wpisywać do przestrzeni krotek informację o tym, jak i o kierunku zmiany. Licznik ma za zadanie pobierać z przestrzeni informacje o zmianie stanu określonego pinu, zliczać liczbę przejść z 0 na 1 i z 1 na 0 dla tego pinu, i cyklicznie wypisywać zebrane informacje na ekran.
Swoisty "szablon" krotki mógłbym wyglądać następująco:

1. nazwa: informacja o kierunku zmiany stanu pinu: "0->1" lub "1->0",

2. pole 1: numer pinu GPIO

3. pole 2: liczba milisekund, które upłynęły od startu procesu

Dokładniejszy opis aplikacji:

1. Proces `sensor`: cyklicznie sprawdza stan określonego pinu GPIO (sensor1: pinu D6, sensor2: pinu D7). Jeśli wykryje zmianę stanu pinu: wpisuje do przestrzeni krotek (funkcja "out") krotkę o konstrukcji opisanej powyżej, z odpowiednimi parametrami (np. 6 jako numer pinu dla procesu sensor1 oraz liczbę milisekund od startu programu (za pomocą funkcji ZsutMillis())).

2. Proces `licznik`: próbuje "wyciągnąć" z przestrzeni krotek informację o zmianie pinu za pomocą szablonu krotki opisanego powyżej (funkcja "inp" - nieblokująca, ponieważ może nie być określonej krotki, a nie chcemy blokować wątku czekając na krotkę, która może nie przyjść). Okresowo także wypisuje te informacje na terminal (u nas: co 4 "odebrane" krotki).

Każdy z sensorów posiada też pliki środowiskowe "infile.txt". Zostały one tak przygotowane, żeby dla sensora 1 aktywować co jakiś czas pin D6, a dla sensora 2 - pin D7. Znajdują się one w podfolderach każdego z tych "urządzeń".
Działanie tej aplikacji zostało przedstawione w filmiku `app2_-_sensors-counter.mkv`, który znajduje się w repozytorium wraz z resztą projektu.