PSIR – Projekt

Michał Drętkiewicz
Eryk Głąb
Patryk Kosiński

13.01.2024

Spis treści

1 Protokół aplikacyjny

Implementacja protokołu zawarta jest w plikach `protocol.h` i `protocol.h`.

Wiadomości protokołu mają następującą strukturę:

```
typedef struct Message {
    uint32_t id;
    MessageType type;
    MessageData data;
} Message;
```

Znaczenie pól:

- **`id`** – jest to unikalny w zakresie każdego połączenie identyfikator wiadomości, wykorzystywany głównie w mechanizmie potwierdzania odbioru (ACK). Kolejne wartości są generowane funkcją `message_next_id()`.
- **`type`** – rodzaj wiadomości. Definicja typu `MessageType`:

  ```
  typedef enum MessageType {
      message_ack,
      message_tuple_space_insert_request,
      message_tuple_space_get_request,
      message_tuple_space_get_reply,
  } MessageType;
  ```

  Znaczenie wartości:

  - **`message_ack`** – potwierdzenie odebrania wcześniejszej wiadomości, przesyłane w obu kierunkach,
  - **`message_tuple_space_insert_request`** – żądanie wstawienia krotki do przestrzeni krotek, wysyłane od klienta do serwera,
  - **`message_tuple_space_get_request`** – żądanie odczytania krotki z przestrzeni krotek, wysyłane od klienta do serwera,
  - **`message_tuple_space_get_reply`** – odpowiedź na wiadomość typu `message_tuple_space_get_request`, przesyłana od serwera do klienta.
- **`data`** – reszta danych wiadomości, zależna od jej typu. Definicja typu `MessageData`:

  ```
  typedef union MessageData {
      MessageAck ack;
      MessageTupleSpaceInsertRequest tuple_space_insert_request;
      MessageTupleSpaceGetRequest tuple_space_get_request;
      MessageTupleSpaceGetReply tuple_space_get_reply;
  } MessageData;
  ```

  Poszczególny pola unii odpowiadają kolejno wartościom `MessageType` opisanym w poprzednim punkcie.

  Definicje typów pól:

  - ```
    typedef struct MessageAck {
        uint32_t message_id;
    } MessageAck;
    ```

    Pole **`message_id`** to `id` wiadomości, której dotyczy to potwierdzenie.

  -
    ```

```
typedef struct MessageTupleSpaceInsertRequest {
    Tuple tuple;
} MessageTupleSpaceInsertRequest;
```

Znaczenie pola takie samo, jak w argumencie funkcji `tuple_space_insert()`, opisanej w następnym rozdziale.

- ```
  typedef struct MessageTupleSpaceGetRequest {
      Tuple tuple_template;
      TupleSpaceOperationBlockingMode blocking_mode;
      TupleSpaceOperationRemovePolicy remove_policy;
  } MessageTupleSpaceGetRequest;
  ```

  Znaczenie pól takie samo, jak w argumentach funkcji `tuple_space_get()`, opisanej w następnym rozdziale.

  ```
  typedef struct MessageTupleSpaceGetReply {
      TupleSpaceOperationResult result;
  } MessageTupleSpaceGetReply;
  ```

  Znaczenie pola takie samo, jak w wartości zwracanej z funkcji `tuple_space_get()`, opisanej w następnym rozdziale.

Podczas transmisji, każda wiadomość poprzedzona jest swoją długością w bajtach, rzutowaną na typ `uint32_t`.
  ```

2 Przestrzeń krotek

Tuple Space definiuje następujące API, zaimplementowane w plikach `tuple_space.h` i `tuple_space.c`:

- Struktura utrzymująca stan przestrzeni krotek:

```
typedef struct TupleSpace {
    size_t tuple_count;
    Tuple* tuples;
} TupleSpace;
```

- Konstruktor pustej przestrzeni krotek:

```
TupleSpace tuple_space_new();
```

- Destruktor przestrzeni krotek, zwalniający zasoby wszystkich przechowywanych.

```
void tuple_space_free(TupleSpace tuple_space);
```

- Metoda wstawiająca krotkę `tuple` do przestrzeni krotek. Nieblokująca. Nie ma wymagania, aby wstawiane krotki były różne.

```
void tuple_space_insert(TupleSpace* tuple_space, Tuple tuple);
```

- Metoda pozyskująca z przestrzeni krotek krotkę pasującą do wzorca `tuple_template`. Jeśli pasuje więcej niż jedna, zwrócona będzie arbitralnie wybrana z nich.

```
TupleSpaceOperationResult tuple_space_get(
    TupleSpace* tuple_space,
    Tuple const* tuple_template,
    TupleSpaceOperationBlockingMode blocking_mode,
    TupleSpaceOperationRemovePolicy remove_policy
);
```

  Parametr `blocking_mode` określa, czy funkcja zablokuje wywołujący ją wątek, gdy w przestrzeni nie będzie krotki pasującej do wzorca. Zdefiniowane są następujące wartości, które oznaczają odpowiednio operację blokująca i nieblokującą:

```
typedef enum TupleSpaceOperationBlockingMode {
    tuple_space_blocking,
    tuple_space_nonblocking,
} TupleSpaceOperationBlockingMode;
```

  Parametr `remove_policy` określa czy w przypadku udanej operacji wybrana krotka ma zostać usunięta z przestrzeni krotek. Zdefiniowane są następujące wartości, które kolejno odpowiadają usunięciu krotki z przestrzeni i pozostawieniu jej:

```
typedef enum TupleSpaceOperationRemovePolicy {
    tuple_space_remove,
    tuple_space_keep,
} TupleSpaceOperationRemovePolicy;
```

  Metoda `tuple_space_get()` zwraca obiekt typu `TupleSpaceOperationResult`:

```
typedef struct TupleSpaceOperationResult {
    TupleSpaceOperationStatus status;
    Tuple tuple;
} TupleSpaceOperationResult;
```

  Pole `status` oznacza odpowiednio powodzenie lub niepowodzenie operacji. W przypadku operacji blokującej, będzie to zawsze `tuple_space_success`, ponieważ metoda nie zakończy

się, aż w przestrzeni będzie krotka pasująca do wzorca. W przypadku operacji nieblokującej, w sytuacji, gdy w przestrzeni nie ma odpowiedniej krotki, pole to ustawiane jest na `tuple_space_failure`.

```
typedef enum TupleSpaceOperationStatus {
    tuple_space_success,
    tuple_space_failure,
} TupleSpaceOperationStatus;
```

Pole `tuple` to wynikowa krotka. W przypadku niepowodzenia operacji nieblokującej będzie to zawsze pusta (0-elementowa) krotka.

- Metoda konwertująca `TupleSpace` do postaci tekstowej:

```
char const* tuple_space_to_string(TupleSpace const* tuple_space);
```

Wynikowy string jest poprawny tylko do momentu kolejnego wywołania tej metody z tego samego wątku. Aby zachować go na dłużej należy wykonać kopię.

Odpowiedniość pomiędzy API z prezentacji do projektu a zaimplementowanym przez nas jest następująca (dla przykładowych parametrów):

- Nieblokujące wstawianie:

```
out(tuple);
tuple_space_insert(&tuple_space, tuple);
```

- Nieblokujący, nieusuwający odczyt:

```
rdp(template);
tuple_space_get(&tuple_space, &template, tuple_space_nonblocking,
tuple_space_keep);
```

- Nieblokujący, usuwający odczyt:

```
inp(template);
tuple_space_get(&tuple_space, &template, tuple_space_nonblocking,
tuple_space_remove);
```

- Blokujący, nieusuwający odczyt:

```
rd(template);
tuple_space_get(&tuple_space, &template, tuple_space_blocking, tuple_space_keep);
```

- Blokujący, usuwający odczyt:

```
in(template);
tuple_space_get(&tuple_space, &template, tuple_space_blocking, tuple_space_remove);
```

Struktura `TupleSpace` i jej wszystkie metody są thread-safe (można je wykonywać współbieżnie), ponieważ według pierwotnego pomysłu serwer miał być zaimplementowany wielowątkowo, co jednak nie okazało się konieczne.

Aplikacje klienckie komunikują się z przestrzenią krotek poprzez serwer (opisany w następnym rozdziale) przy pomocy protokołu aplikacyjnego (opisanego w poprzednim rozdziale).

3 Serwer

Serwer oczekuje na połączenie klientów nasłuchując na określonym z góry porcie UDP na wszystkich interfejsach. W głównej pętli programu oczekujemy na wiadomości od klientów i przetwarzamy je zależnie od ich typu, modyfikując przestrzeń krotek i/lub odsyłając odpowiedzi do klientów.

```
for(;;) {
    InboundMessage inbound_message = network_receive_message_blocking(
        &server->network
    );
    server_handle_inbound_message_nonblocking(server, inbound_message);
    server_process_blocked_get_requests(server);
}
```

W osobny sposób traktujemy blokujące żądania odczytu krotki wysłane przez klientów. Przy odebraniu takiego żadania sprawdzamy, czy może ono być spełnione od razu, i jeśli tak, to odsyłamy odpowiedź. W przeciwnym razie zapisujemy żądanie do tablicy `blocked_get_requests` struktury `Server` i nie wysyłamy żadnej odpowiedzi do klienta, tylko przechodzimy do obsługi następnych przychodzących wiadomości.

Po przetworzeniu każdej odebranej wiadomości od dowolnego z klientów wracamy do tablicy `blocked_get_requests` i sprawdzamy które z zapytań w niej zawartych mogą być spełnione od razu i tylko na nie odsyłamy odpowiedzi. Korzystamy tu z obserwacji, że jedynym zdarzeniem, które mogłoby odblokować oczekujące żądanie jest przyjście wiadomości typu `message_tuple_space_insert_request` od któregoś z klientów. Ta strategia pozwala na to, aby serwer był współbieżny i jednocześnie jednowątkowy, co upraszcza w pewnym stopniu implementację.

Jedyną niestandardową strukturą danych wykorzystaną w implementacji serwera jest wektor, czyli dynamicznie alokowana tablica zmiennej wielkości, stosowana między innymi do przechowywania blokujących wiadomości typu `message_tuple_space_get_request`, buforów na dane przychodzące z sieci, itp. Ponieważ często wykorzystywaną operacją jest usuwanie elementów z dowolnej pozycji wektora, jego elementy nie gwarantują stałej kolejności, dzięki czemu usuwanie elementu implementowane jest jako nadpisanie go ostatnim elementem i zmniejszenie długości wektora o 1, co daje stałą złożoność obliczeniową.

4 Aplikacja 1.

Aplikacja składa się z dwóch procesów: **master** (`main-app1-master.cpp`), oraz **worker** (`main-app1-worker.cpp`). Master produkuje pewną liczbę zadań polegających na sprawdzeniu, czy liczba jest pierwsza, generując krotki postaci `("app1", "is prime", n)` dla `n` z pewnego zakresu liczba naturalnych i wysyłając je do serwera.

Workery w pętli blokująco czekają na powyższe krotki, a gdy ich zapytanie zostanie obsłużone, pobierają i usuwają krotkę z przestrzeni krotek i wyznaczają pierwszość zadanej liczby. Po pewnym opóźnieniu (symulującym bardziej złożone obliczenia zajmujące pewną ilość czasu), krotka wynikowa postaci `("app1", "prime", n)` lub `("app1", "not prime", n)` jest odsyłana do przestrzeni krotek.

Po wygenerowaniu wszystkich zadań master w pętli odpytuje serwer o krotki wynikowe, używając wzoru `("app1", "prime", int?)` i `("app1", "prime", int?)` aż do momentu, w którym zbierze ich tyle ile zadań zostało wygenerowanych.

Gdy jednocześnie działa więcej niż jeden worker, możemy zaobserwować automatyczny load-balancing, ponieważ procesy te pobierają nowe zadania wyłącznie, gdy nie są zajęte obliczaniem wcześniejszego.

Implementacja mastera jest na funkcje:

- generacja nowego zadania dla liczby `n`, czyli stworzenie i wstawienie krotki do przestrzeni krotek za pośrednictwem wiadomości do serwera:

  ```
  void create_task(
      Network* network,
      ArduinoNetworkAddress server_address,
      int32_t number
  )
  ```

- stworzenie i wysłanie do serwera zapytania o krotki wynikowe, gdzie parametr `query` przyjmuje wartości `"prime"` lub `"not prime"`, zgodnie z tym, co generują workery.

  ```
  void send_query_message(
      Network* network,
      ArduinoNetworkAddress server_address,
      char const* query
  )
  ```

- inicjalizacja połączenia z serwerem, zlecenie generowania zadań i zbierania wyników, oraz zwolnienie zasobów po zakończeniu:

  ```
  void setup();
  ```

Przykład wykonania (niektóre linie zostały wycięte i zastąpione `...` dla skrócenia wypisu):

- master:

  ```
  [133] New task: ("app1", "is prime", 0)
  [155] New task: ("app1", "is prime", 1)
  ...
  [1211] New task: ("app1", "is prime", 30)
  [1248] New task: ("app1", "is prime", 31)
  [3078] Reply: ("app1", "not prime", 0)
  [4940] Reply: ("app1", "prime", 31)
  [4980] Reply: ("app1", "not prime", 1)
  ...
  ```

```
[58864] Reply: ("app1", "prime", 5)
[60628] Reply: ("app1", "prime", 3)
[64062] Reply: ("app1", "not prime", 4)
[64085] Done
```

- worker:

```
[2161] Accepting task: ("app1", "is prime", 0)
[3683] Sending reply: ("app1", "not prime", 0)
[3720] Accepting task: ("app1", "is prime", 1)
[5897] Sending reply: ("app1", "not prime", 1)
[5933] Accepting task: ("app1", "is prime", 31)
[6085] Sending reply: ("app1", "prime", 31)
...
```

5 Aplikacja 2.

6 Environment file dla aplikacji 2.