

PSIR – Projekt

Michał Drętkiewicz

Eryk Głąb

Patryk Kosiński

13.01.2024

Spis treści

1 Protokół aplikacyjny	3
2 Przestrzeń krotek	4
3 Serwer	6
4 Aplikacja 1.	7
5 Aplikacja 2.	7
6 Environment file dla aplikacji 2.	7

1 Protokół aplikacyjny

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

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 połączenia 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;
```

Poszczególne 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 argumentcie 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 blokującą 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, nieusuwaający 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 żądania 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.

5 Aplikacja 2.

6 Environment file dla aplikacji 2.