IPC Publish-Subscribe

Anna Zalesińska, 155868
Mateusz Juszczak, 155968

Link do repozytorium: IPC-Publish-Subscribe

1 Protokół komunkacji

Cała komunikacja klient-serwer odbywa się za pomocą struktury message, zawierającą wszystkie pola potrzebne do efektywnej komunikacji.

```
struct message {
        long mtype;
        int id;
        int cnt;
        char username[USERNAME_LEN];
        char topicname[TOPIC_LEN];
        char topic_id;
        int sub_duration;
        char sub_topic[20];
        char text[100];
        char topic_list[TOPIC_LEN*TOPIC_CNT];
};
```

Poszczególne komunikaty mają przydzielone kody znajdujące się w pliku nagłówkowym. Kody są jednocześnie priorytetami typów wiadomości oraz ich identyfikatorami. Obsługa komunikatów typu "żadanie klienta"(żadanie wykonania operacji takich jak np. wysłanie listy aktywnych tematów, dodanie do listy subskrybentów, zablokowanie użytkownika; są to komendy rozpoczynające się od znaku "/"oraz procedura logowania) odbywa się w sposób synchroniczy. Po nadaniu komunikatu żądania, klient jest blokowany do momentu otrzymania odpowiedzi. Inaczej to działa w przypadku wysyłania wiadomości tekstowych. Odbywa się to asynchronicznie, czyli klient po nadaniu wiadomości wraca do dalszego przetwarzania. W cyklu pracy klienta jest również etap odbierania wiadomości tekstowych oraz komunikatów serwerowych z kolejki. Serwer przetwarza komunikaty synchronicznie, blokując się na wywołaniu fukncji msgrcv w przypadku braku wiadomości do przetworzenia.

1.1 Logowanie i rejestracja obiorcy

Procedura logowania rozpoczyna się zaraz po uruchomieniu programu. Użytkownik wybiera unikalny login, który będzie jego identyfikatorem publicznym dla innych użytkowników. Niezależnie od użytkownika, program pobiera systemowy identyfikator procesu, na podstawie którego będzie generowny klucz kolejki klienta.

```
int send_login(int client_q , int server_q, const char* username) {
  struct message loginMessage;
  loginMessage.mtype = CR_LOGIN;
  loginMessage.id = client_q;
  strcpy(loginMessage.username, username);
  msgsnd( server_q, &loginMessage,
          sizeof(struct message) - sizeof(long), 0);
  msgrcv(client_q, &loginMessage,
          sizeof(struct message) - sizeof(long), CR_LOGIN, 0);
  return loginMessage.id == SR_ERR;
}
```

Serwer identyfikuje typ wiadomości zestawiając zawartość pola mtype z kodami z pliku parameters.h . Następnie weryfikuje unikalność nazwy użytkownika oraz dodaje go do domyślnego tematu serwera.

```
case CR_LOGIN:
  response.mtype = CR_LOGIN;
  if(find_client_by_username(active_clients, msg.username) != NULL) {
    strcpy(response.text, "username taken");
    response.id = SR_ERR;
  } else {
    if(find_topic_by_name(active_topics, DEFAULT_TOPIC) == NULL) {
      add_topic(active_topics, DEFAULT_TOPIC, active_clients, -1);
      printTopicsAndSubscribers(active_topics);
    }
    add_client(active_clients, msg.id, msg.username);
    add_modify_sub( active_topics, DEFAULT_TOPIC,
                    active_clients, -1, msg.id);
    response.id = SR_OK;
  }
break;
```

1.2 Rejestracja subskrypcji

Klient w dowolnym momencie może zażądać listy aktywnych tematów używając komendy "/topiclist", a następnie zasubskrybować czasowo lub trwale wybrany temat używając komendy "/sub *temat* [*ile wiadomości*]".

```
int subscribe_modify( int client_q , int server_q,
                      const char* topicname, int duration) {
  struct message subMessage;
  subMessage.mtype = CR_ADD_SUB;
  subMessage.id = client_q;
  subMessage.cnt = duration;
  strcpy(subMessage.topicname, topicname);
  msgsnd( server_q, &subMessage,
          sizeof(struct message) - sizeof(long), 0);
  msgrcv( client_q, &subMessage,
          sizeof(struct message) - sizeof(long), CR_ADD_SUB, 0);
  return subMessage.id == SR_ERR;
}
```

```
case CR_ADD_SUB:
  response.mtype = CR_ADD_SUB;
  struct Topic* topic = find_topic_by_id(active_topics, msg.topic_id);
  add_modify_sub( active_topics, topic->name,
                  active_clients, msg.cnt, msg.id);
  response.mtype = SR_OK;
  send_info_about_new_topic(active_clients, msg.topicname, msg.id);
  printTopicsAndSubscribers(active_topics);
break;
```

1.3 Rejestracja typu wiadomości (tematu)

Klient w dowolnym momencie może zażądać rejestracji nowego (dotychczas nieistniejącego) tematu. W tym celu może użyć komendy "/newtopic *temat*". Gdy temat zostanie utworzony, klient zostanie automatycznie dodany do listy subskrybentów, a wszyscy użytkownicy zostaną poinformowaniu o utworzeniu nowego tematu. Klient może swobodnie wybierać temat, na który aktualnie chce pisać używając komendy "/topic *temat*".

```
int create_topic(int client_q , int server_q, const char* topicname) {
  struct message topicMessage;
  topicMessage.mtype = CR_CREAT_TOPIC;
  topicMessage.id = client_q;
  strcpy(topicMessage.topicname, topicname);
  msgsnd( server_q, &topicMessage,
          sizeof(struct message) - sizeof(long), 0);
  msgrcv( client_q, &topicMessage,
```

```
            sizeof(struct message) - sizeof(long), CR_CREAT_TOPIC, 0);
  return topicMessage.id == SR_ERR;
}

case CR_ADD_SUB:
  response.mtype = CR_ADD_SUB;
  struct Topic* topic = find_topic_by_id(active_topics, msg.topic_id);
  add_modify_sub(active_topics, topic->name,
                 active_clients, msg.cnt, msg.id);
  response.mtype = SR_OK;
  send_info_about_new_topic(active_clients, msg.topicname, msg.id);
  printTopicsAndSubscribers(active_topics);
break;
```

1.4 Rozgłaszanie nowej wiadomości

```
int send_message( int server_q, int client_q ,
                  char* topic, char* username, char* text) {
  struct message textMessage;
  textMessage.mtype = CR_TEXTMSG;
  textMessage.id = client_q;
  strcpy(textMessage.topicname, topic);
  strcpy(textMessage.username, username);
  strcpy(textMessage.text, text);
  return messageSend(server_q, &textMessage);
}

case CR_TEXTMSG:
  struct Topic* topic1 = find_topic_by_name(active_topics, msg.topicname);
  struct Sub* sub = topic1->subscribers->head;
  strcpy(response.topicname, msg.topicname);
  strcpy(response.username, msg.username);
  strcpy(response.text, msg.text);
  response.mtype = SR_TEXTMSG;
  response.id = SR_OK;
  while(sub != NULL){
  if(find_blocked_by_id(sub->client->blocked, msg.id) == NULL){
    msgsnd( sub->client->id, &response,
            sizeof(struct message) - sizeof(long), 0);
  }
  sub = sub->next;
}
continue;
break;
```

1.5 Odbiór wiadomości w sposób asynchroniczny

Klient w toku przetwarzania sprawdza, czy w jego kolejce znajduje się wiadomość. W ten sposób odbiera on wszystkie komunikaty serwerowe oraz wiadomości tekstowe przychodzące od serwera. W tym celu zastosowaliśmy ujemny argument priorytetu, aby uniknąć sprawdzania wszystkich priorytetów osobno. Swoje zastosowanie znajduje tutaj również flaga IPC_NOWAIT, która umożliwia natychmiastowe wznowienie przetwarzania w przypadku braku pracy.

```
if (msgrcv(client_q, &msg, sizeof(struct message),
    -CR_TEXTMSG, IPC_NOWAIT) != -1) {
  char* header =
  (char*) malloc(sizeof(char) * (USERNAME_LEN + 1 + TOPIC_LEN));
  strcpy(header, msg.username);
  strcat(header, "@");
  strcat(header, msg.topicname);
  addMessageToBuffer( messageLogBuffer,
                      createMessageEntry(header, msg.text));
  free(header);
}
```

1.6 Blokowanie wiadomości od użytkownika

Klient w dowolnym momencie może zażądać zablokowania dowolnego użytkownika. W tym celu może użyć komendy "/mute *username*". Od tego momentu nie będzie on otrzymywał od serwera wiadomości nadanych przez tego użytkownika oraz nie będą one wpływały na licznik subskrybcji czasowej.

```
int block_user(int server_q, int client_q, char* username) {
  struct message muteMessage;
  muteMessage.mtype = CR_MUTE;
  muteMessage.id = client_q;
  strcpy(muteMessage.username, username);
  messageSend(server_q, &muteMessage);
  messageReceive(client_q, &muteMessage, CR_MUTE);
  return muteMessage.id == SR_ERR;
}
```

```
case CR_MUTE:
response.mtype = CR_MUTE;
struct Client* client = find_client_by_id(active_clients, msg.id);
struct Client* to_mute =
        find_client_by_username(active_clients, msg.username);
if(to_mute == NULL) {
  strcpy(response.text, "no such user");
} else {
  add_blocked(client->blocked, to_mute);
```

```
    displayBlockedList(client);
    response.id = SR_OK;
  }
  break;
```