Programowanie Sieciowe

Serwer do obsługi usługi IRC

Z51. Dokumentacja końcowa

Maciej Bogusławski (331362)
Hubert Kaczyński (331386)
Bartosz Żelazko (331457)

Spis treści

1. Zedytowana dokumentacja projektu wstępnego

Zmienione elementy względem oryginalnego planu projektu wstępnego

- W pierwszym przypadku użycia poprawiono nazwę błędu na poprawną "ERR_NICKNAMEINUSE"
- Po głębszym poznaniu tematyki IRC przyjęto założenie, że nick może ulec zmianie po rejestracji, więc usunięto sytuację błędną "1.6. Użytkownik ponownie wysyła NICK po rejestracji".
- Usunięto z planowanej implementacji komendę KILL, ponieważ jej zastosowanie wykracza poza zakres uproszczonej implementacji serwera IRC.
- Zdecydowano o otestowaniu działania programu przy użyciu google test, który korzystając z biblioteki klienckiej komunikują się z instancją serwera, testując używane komendy i komunikaty.
- Dodano implementacje komend CAP i TOPIC wraz z obsługą błędów.
- W celu umożliwienia korzystania z serwera wielu klientom zdecydowano się na oparciu architektury na użyciu epoll.

Powyższe zmiany zostały uwzględnione w zedytowanej wersji projektu wstępnego w poniższym rozdziale.

Treść zadania

Celem projektu jest implementacja uproszczonego serwera IRC w języku C++. Serwer powinien obsługiwać uwierzytelnianie użytkowników, komunikację między pojedynczymi użytkownikami w formie wiadomości prywatnych, komunikację w kanałach grupowych, zarządzanie kanałami (poprzez tworzenie, dołączenie do kanałów lub opuszczanie kanałów) oraz administrację (usuwanie użytkowników z kanału przez operatora). Zaimplementowany serwer powinien umożliwiać obsługę wielu klientów jednocześnie. Implementacja projektu oparta jest na specyfikacji RFC 1459 z uproszczeniami dostosowanymi do zakresu i założeń czasowych projektu.

Niniejszy dokument stanowi realizację etapu drugiego (projekt wstępny) zadania projektowego. Przedstawia on m.in. fundamentalne założenia projektowe, przypadki użycia oraz opis architektury, wybranego środowiska oraz użytych narzędzi. Przedstawione w ramach dokumentacji informacje stanowią wstępną wizję implementacji i mogą ulec zmianie w trakcie dalszego rozwoju projektu.

Założenia funkcjonalne

1. Serwer korzysta z protokołu TCP do komunikacji,
2. Serwer umożliwia na jednoczesne połączenie większej ilości klientów niż jeden,
3. Serwer oraz klient używają ustandaryzowanych komend do komunikacji,
4. Serwer na czas działania programu przechowuje nick, hasło i username połączonych klientów,
5. Serwer umożliwia wylistowanie na prośbę klienta wszystkich dostępnych do komunikacji klientów i kanałów,
6. Serwer umożliwia przesyłanie wiadomości prywatnych między dwoma klientami,
7. Serwer umożliwia utworzenie nowego kanału do komunikacji przez klienta,
8. Serwer umożliwia klientom dołączenie do wybranego istniejącego kanału,
9. Serwer nadaje prawa operatora kanału klientowi tworzącemu kanał,
10. Serwer umożliwia użytkownikom z prawami operatora kanału usuwanie innych użytkowników z operowanego kanału oraz zmianę tematyki kanału,
11. Serwer umożliwia klientom samodzielne opuszczenie kanałów,
12. Serwer umożliwia klientom połączonym z kanałem wysyłanie wiadomości widocznych dla wszystkich połączonych z kanałem,
13. Serwer może samodzielnie zamknąć połączenie z klientem,

14. Serwer umożliwia wysłanie komunikatu ping do wybranego klienta i odbiór komunikatu pong od klienta.

Założenia niefunkcjonalne

1. Zarządzanie środowiska jest możliwe z użyciem make,
2. Serwer zawiera obsługę błędów,
3. Serwer jest odporny na błędne lub zmanipulowane komunikaty,
4. Testy automatyczne serwera uwzględniają obsługę błędów,
5. Wszystkie komponenty serwera generują czytelne logi,
6. Program prawidłowo obsługuje sygnały SIGINT oraz SIGTERM,
7. Program serwera posiada wysokie pokrycie kodu testami,
8. Środowisko wykorzystane do prezentacji działania serwera nie wymaga dodatkowej konfiguracji.

Przykładowe przypadki użycia

1. Otwarcie połączenia klient- serwer.

W celu dołączenia do serwera IRC klient na samym początku wysyła do serwera wiadomość PASS podając przy tym hasło do serwera. Jeśli hasło było poprawne połączenie zostanie utworzone. W przeciwnym przypadku otrzyma on błąd ERR_ALREADYREGISTRED. Po udanym wprowadzeniu hasła użytkownik musi wysłać wiadomość NICK w której podaje swoją nazwę użytkownika. Nick musi być oryginalny, w przeciwnym przypadku serwer odpowie błędem ERR_NICKNAMEINUSE. Ostatnim krokiem do ukończenia rejestracji jest wysłanie wiadomości USER. W przypadku gdy użytkownik jest już zarejestrowany na wiadomość USER serwer powinien zwrócić błąd ERR_ALREADYREGISTRED.

2. Wysłanie wiadomości pomiędzy dwoma klientami

Aby zalogowany klient mógł wysłać do innego zalogowanego klienta wiadomość musi on wysłać wiadomość PRIVMSG, gdzie jako pierwszy argument podany zostanie nick klienta do którego wiadomosc ma zostać wysłana. Na wiadomość tą serwer może odpowiedzieć błędem ERR_NOSUCHNICK gdy nie można znaleźć użytkownika o danym nicku. Dodatkowo przed wysłaniem wiadomości możliwe jest wyszukanie użytkownika przy użyciu wiadomości WHO, która pozwala wypisać wszystkich użytkowników lub wyszukać konkretnego użytkownika.

3. Stworzenie kanału

W celu stworzenia kanału należy wysłać do serwera wiadomość JOIN z unikalną nazwą kanału. To spowoduje utworzenie nowego kanału o nazwie podanej jako parametr. Użytkownik który tworzy kanał automatycznie staje się jego operatorem.

4. Dołączenie do kanału

W celu dołączenia do kanału należy wysłać wiadomość JOIN z nazwą kanału do którego użytkownik chce dołączyć.

5. Wysłanie wiadomości do kanału

Wysłanie wiadomości do kanału działa analogicznie do wysłania wiadomości do pojedynczego użytkownika. Jako adresat podawana jest jednak nie nick klienta, a nazwa wybranego kanału.

6. Opuszczenie kanału

Kanał można opuścić poprzez wysłanie wiadomości PART, której argumentami są nazwy kanałów, które użytkownik chce opuścić. Wiadomość ta może generować błędy ERR_NOSUCHCHANNEL gdy podano nieistniejącą nazwę kanału lub błąd ERR_NOTONCHANNEL gdy użytkownik nie jest członkiem kanału, który chce opuścić.

7. Usunięcie użytkownika z kanału

Operator może usunąć użytkownika z kanału poprzez wiadomość KICK i podanie nazwy kanału i użytkownika, który ma zostać usunięty. Wiadomość ta może spowodować zwrócenie przez serwer błędu ERR_CHANOPRIVSNEEDED gdy wysyłający nie ma uprawnień lub błąd ERR_NOSUCHCHANNEL jeśli podana zostanie błędna nazwa kanału. Serwer natomiast zwrócić powinien błąd ERR_NOTONCHANNEL gdy użytkownik, który ma zostać usunięty z kanału się w nim nie znajduje.

8. Zakończenie połączenia

Zakończenie połączenia umożliwia wiadomość QUIT.

Analiza i obsługa sytuacji błędnych

Poniżej znajdują się przykładowe sytuacje błędne.

Błędy podczas procesu rejestracji

1.1. Nieprawidłowe hasło klienta

Klient podaje hasło, które nie może zostać zatwierdzone.\
Wywołanie: `PASS <password>`\
Obsługa: Odpowiedź: ERR_ALREADYREGISTRED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.2. Za mało argumentów komunikatu hasła

Klient podaje za mało argumentów wywołania\
Wywołanie: `PASS <password>`\
Obsługa: Odpowiedź: ERR_NEEDMOREPARAMS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.3. Użytkownik ponownie wysyła PASS po rejestracji

Klient wysyła komunikat PASS po ówcześnie udanym użyciu PASS\
Wywołanie: `PASS <password>`\

Obsługa: Odpowiedź: ERR_ALREADYREGISTRED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.4. Nieprawidłowy nick klienta

Klient podaje nickname, który nie może zostać zatwierdzony.\
Wywołanie: `NICK <nickname>`\
Obsługa: Odpowiedź: ERR_ERRONEUSNICKNAME\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.5. Za mało argumentów NICK

Klient podaje za mało argumentów wywołania\
Wywołanie: `NICK <nickname>`\
Obsługa: Odpowiedź: ERR_NONICKNAMEGIVEN\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.6. Użytkownik podał używany już przez kogoś nickname

Klient wysyła argument nickname identyczny do innego aktualnie połączonego klienta\
Wywołanie: `NICK <nickname>`\
Obsługa: Odpowiedź: ERR_NICKNAMEINUSE\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.7. Nieprawidłowy username klienta

Klient podaje username, który nie może zostać zatwierdzony.\
Wywołanie: `USER <username> <hostname> <servername> <realname>`\
Obsługa: Odpowiedź: ERR_ALREADYREGISTRED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.8. Za mało argumentów USER

Klient podaje za mało argumentów wywołania\
Wywołanie: `USER <username> <hostname> <servername> <realname>`\
Obsługa: Odpowiedź: ERR_NEEDMOREPARAMS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.9. Użytkownik ponownie wysyła USER po rejestracji

Klient wysyła komunikat USER po ówcześnie udanym użyciu USER\
Wywołanie: `USER <username> <hostname> <servername> <realname>`\
Obsługa: Odpowiedź: ERR_ALREADYREGISTRED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

1.10. Użytkownik podał używany już przez kogoś username

Klient wysyła argument username identyczny do innego aktualnie połączonego klienta\

Wywołanie: `USER <username> <hostname> <servername> <realname>`\
Obsługa: Odpowiedź: ERR_ALREADYREGISTRED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

Błędy dotyczące komunikacji użytkownika

2.1. Wysyłanie komend przez niezarejestrowanego użytkownika

Klient wysyła komendę inną niż rejestracja bez wcześniejszej rejestracji.\
Obsługa: Odpowiedź: ERR_NOTREGISTERED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

2.2. Wiadomość do nieistniejącego użytkownika

Klient wysyła wiadomość podając adresata który nie jest połączony z serwerem\
Wywołanie: `PRIVMSG <receiver> <text>`\
Obsługa: Odpowiedź: ERR_NOSUCHNICK\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

2.3. Próba wysłania pustej wiadomości (brak tekstu)

Klient nie podał argumentu text\
Wywołanie: `PRIVMSG <receiver> <text>`\
Obsługa: Odpowiedź: ERR_NOTEXTTOSEND\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

2.4. Próba wysłania wiadomości podając za dużo adresatów

Klient podał zbyt dużo argumentów (może być tylko jeden adresat)\
Wywołanie: `PRIVMSG <receiver> <text>`\
Obsługa: Odpowiedź: ERR_TOOMANYTARGETS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

2.5. Próba wysłania wiadomości bez podania odbiorcy

Argument receiver nie został podany\
Wywołanie: `PRIVMSG <receiver> <text>`\
Obsługa: Odpowiedź: ERR_NORECIPIENT\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

Błędy związane z kanałami

3.1. Próba dołączenia do kanału, do którego klient już należy

Klient podał nieistniejący kanał w argumencie JOIN\
Wywołanie: `JOIN <channel>`\
Obsługa: Odpowiedź: ERR_NOSUCHCHANNEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.2. Próba dołączenia do kanału bez podania argumentu

Klient nie podał argumentu channel\
Wywołanie: `JOIN <channel>`\
Obsługa: Odpowiedź: ERR_NEEDMOREPARAMS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.3. Próba dołączenia do kilku kanałów naraz

Klient podał za dużo argumentów channel\
Wywołanie: `JOIN <channel>`\
Obsługa: Odpowiedź: ERR_TOOMANYCHANNELS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.4. Opuszczanie kanału, który nie istnieje

Klient podał nieistniejący kanał w argumencie PART\
Wywołanie: `PART <channel>`\
Obsługa: Odpowiedź: ERR_NOSUCHCHANNEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.5. Opuszczanie kanału bez podania argumentu

Klient nie podał argumentu channel dla PART\
Wywołanie: `PART <channel>`\
Obsługa: Odpowiedź: ERR_NEEDMOREPARAMS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.6. Opuszczanie kanału, na którym się nie jest

Klient podał kanał, z którym nie jest połączony\
Wywołanie: `PART <channel>`\
Obsługa: Odpowiedź: ERR_NOTONCHANNEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.7. Wysyłanie wiadomości na kanał, na którym klient nie jest obecny

Klient podał kanał, z którym nie jest połączony jako argument\
Wywołanie: `PRIVMSG <receiver> <text>`\
Obsługa: Odpowiedź: ERR_CANNOTSENDTOCHAN\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.8. Użycie KICK na kanale, na którym się nie jest

Klient użył komendy KICK, nie będąc połączonym z podanym kanałem\
Wywołanie: `KICK<channel> <user>`\
Obsługa: Odpowiedź: ERR_NOTONCHANNEL\

Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.9. Użycie KICK bez podania argumentów

Klient użył komendy KICK, nie podając potrzebnych argumentów\
Wywołanie: `KICK<channel> <user>`\
Obsługa: Odpowiedź: ERR_NEEDMOREPARAMS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.10. Użycie KICK bez uprawnień operatora

Klient użył komendy KICK, nie będąc operatorem\
Wywołanie: `KICK<channel> <user>`\
Obsługa: Odpowiedź: ERR_CHANOPRIVSNEEDED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

3.11. Użycie KICK podając nieistniejący kanał

Klient użył komendy KICK, podając nieistniejący kanał jako argument\
Wywołanie: `KICK<channel> <user>`\
Obsługa: Odpowiedź: ERR_NOSUCHCHANNEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

4. Błędy połączeniowe i komunikatowe

4.1. Klient nie odpowiada na PING

Aby testować połączenie z klientem regularnie wysyłany jest mu komunikat PING, na który serwer oczekuje odpowiedzi PONG\
Obsługa: Po minięciu ustalonego timeout wysyłana jest informacja o zamknięciu połączenia
Zamknięcie połączenia

4.2. Odebrano pustą lub niepoprawnie sformatowaną komendę

Komenda została źle wpisana\
Obsługa: ERR_UNKNOWNCOMMAND\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

4.3. Odebrano komendę przekraczającą maksymalną ilość znaków

Otrzymana komenda jest za długa\
Obsługa: ERR_WILDTOPLEVEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

5. Błędy związane z procesem zamykania połączeń przez klienta

5.1. Klient wysyła QUIT bez wiadomości

Klient nie podał argumentu Quit message\
Wywołanie: `QUIT [<Quit message>]`\
Obsługa: Zamknięcie połączenia z domyślną wiadomością "Client Quit"

6. Błędy związane z komendą TOPIC

6.1. Klient wysyła TOPIC bez wymaganych parametrów

Klient nie podał wymaganych argumentów\
Wywołanie: `TOPIC <channel> <topic>`\
Obsługa: ERR_NEEDMOREPARAMS\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

6.2. Klient wysyła TOPIC z niewłasciwą nazwą kanału

Klient źle podał nazwę kanału\
Wywołanie: `TOPIC <channel> <topic>`\
Obsługa: ERR_NOSUCHCHANNEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

6.3. Klient wysyła TOPIC z nazwą nieistniejącego kanału

Klient podał nazwę nieistniejącego kanału\
Wywołanie: `TOPIC <channel> <topic>`\
Obsługa: ERR_NOSUCHCHANNEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

6.4. Klient wysyła TOPIC dla kanału, którego nie jest członkiem

Klient podał nazwę kanału, którego nie jest członkiem\
Wywołanie: `TOPIC <channel> <topic>`\
Obsługa: ERR_NOTONCHANNEL\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

6.5. Klient wysyła TOPIC dla kanału, którego nie jest operatorem

Klient podał nazwę kanału, którego nie jest operatorem\
Wywołanie: `TOPIC <channel> <topic>`\
Obsługa: ERR_CHANOPRIVSNEEDED\
Próba zostaje odrzucona, ale połączenie nie zostaje zerwane.

Środowisko i narzędzia

Środowisko programistyczne

Implementacja rozwiązania projektowego wykonana zostanie w środowisku Linux w języku C++. Proces kontroli wersji realizowany będzie w oparciu o repozytorium dostępne na wydziałowej platformie GitLab.

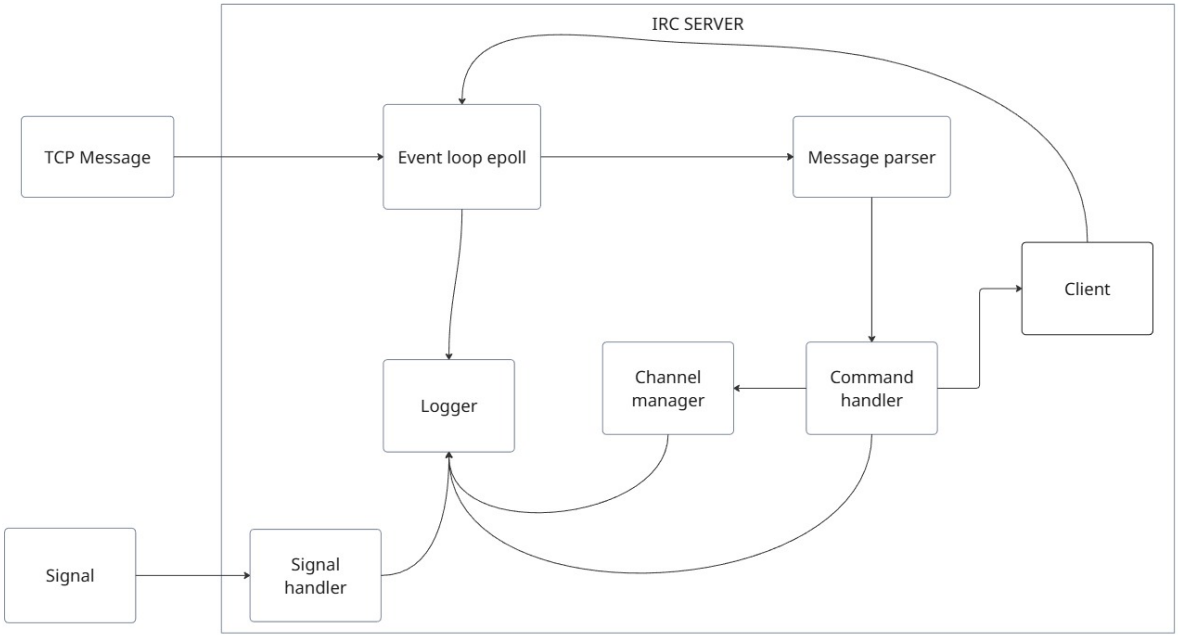
Biblioteki

Logowanie przeprowadzane będzie za pomocą biblioteki spdlog, umożliwiającej filtrowanie poziomów logów za pomocą set_level. Testy przeprowadzane będą za pomocą googletest. Testowanie współdziałania komponentów serwera i klienta umożliwione będzie z wykorzystaniem istniejącej biblioteki klienckiej, takiej jak np. Weechat lub libircclient.

Narzędzia deweloperskie

Zarządzanie środowiskiem umożliwione będzie z użyciem `make`. Za detekcję wycieków pamięci odpowiadać będzie narzędzie diagnostyczne `valgrind`. W celu przeprowadzenia testów końcowych użyta zostanie gotowa istniejąca biblioteka kliencka, np. `Weechat`. Konteneryzacja zapewniająca możliwość prezentacji projektu bez dodatkowej konfiguracji umożliwiona będzie za pomocą narzędzi Docker oraz Docker Compose.

Architektura rozwiązania



Głównym komponentem aplikacji jest klasa Server, która uruchamia jednowątkową pętlę zdarzeń opartą o epoll (tryb edge-triggered) i nieblokujące gniazda. Server nasłuchuje gniazda serwera oraz gniazd klientów; po akceptacji połączenia tworzy obiekt Client, zapisuje go w mapie i dodaje do epoll. Odczyt danych odbywa się w handleClientData: bajty są dokładane do bufora klienta, a processClientMessage wycina linie zakończone CRLF, tokenizuje je przez MessageParser i przekazuje do CommandHandler.

CommandHandler rejestruje i dispatchuje implementacje komend (np. PASS, CAP, NICK, USER, PING/PONG, QUIT, WHO, PRIVMSG, JOIN, PART, KICK, LIST, NAMES, TOPIC). Weryfikuje też wymagania rejestracji/hasła. Stan klienta (nick, username, rejestracja, CAP, kanały, bufor, timery PING/PONG) jest przechowywany w obiekcie Client.

Zarządzaniem kanałami zajmuje się ChannelManager oraz Channel (członkowie, operatorzy, topic). Server cyklicznie realizuje keepalive: wysyła PING i rozłącza klientów bez PONG w określonym czasie. Logowanie realizuje Logger (spdlog, konsola + plik). Signal handlerObsługa SIGINT/SIGTERM w main zatrzymuje pętlę serwera przez Server::stop, a SIGPIPE jest ignorowany.

Lista obsługiwanych komunikatów

- `PASS <password>` - uwierzytelnienie hasłem serwera
- `NICK <nickname>` - ustawienie unikalnego nicku
- `USER <username> <hostname> <servername> <realname>` - rejestracja użytkownika
- `QUIT [<Quit message>]` - rozłączenie (ew. powód rozłączenia)
- `JOIN <channel>` - dołączenie do kanału lub założenie nowego kanału w przypadku gdy kanał o podanej nazwie nie istnieje.
- `PART <channel>` - opuszczenie kanału o podanej nazwie
- `NAMES <channel>` - wyświetlenie listy użytkowników którzy są dostępni na danym serwerze
- `LIST` - wypisuje wszystkie dostępne kanały i użytkowników
- `KICK <channel> <user>` - usunięcie użytkownika z kanału (dostępne dla operatora kanału)
- `PRIVMSG <receiver> <text>` - wysłanie wiadomości na kanał lub do użytkownika
- `WHO <name>` - wyszukanie użytkowników po nazwie lub nazwie częściowej
- `PING` - wiadomość kontrolna sprawdzająca połączenie z użytkownikiem
- `PONG` - odpowiedź na wiadomość ping
- `CAP` - uzgodnienie możliwości między klientem a serweram
- `TOPIC <channel> <topic>` - nadanie tematu kanałowi przez klieta z prawami operatora kanału

Sposób testowania

Wysokie pokrycie kodu testami stanowi fundamentalne założenie naszej realizacji zadania projektowego. Implementacja testów przeprowadzona zostanie z pomocą biblioteki googletest, która zrealizuje szereg testów integracyjnych. Testy obejmować będą m.in. parsowanie wiadomości IRC, zapewniając odporność serwera na błędne lub zmanipulowane komunikaty - serwer powinien poprawnie parsować wiadomości zgodnie z założonymi regułami dotyczącymi m.in. limitów długości, a także odrzucać błędne komunikaty. Testy obejmować będą również m.in. weryfikację walidacji nicków użytkowników oraz nazw tworzonych kanałów, a także poprawności implementacji logiki poszczególnych komend (np. związanych z dodawaniem oraz usuwaniem użytkowników z kanałów).

Testy umożliwiać powinny weryfikację poprawności współdziałania serwera z podłączonymi do niego użytkownikami, prezentując przykładowe realne scenariusze użycia programu z użyciem klas pomocniczych. System uruchomi serwer jako osobny proces, a następnie stworzy klientów TCP z pomocą istniejących bibliotek klienckich. W ramach każdego testu, system wyśle komendy IRC i zweryfikuje odpowiedzi. W ten sposób możliwe stanie się testowanie serwera przez prawdziwe połączenia TCP, dzięki czemu zaimplementowane testy będą mogły faktycznie badać komunikację między wieloma klientami.

Podział pracy w zespole

W celu implementacji każdego elementu projektu - zarówno kodu jak i dokumentacji - zespół spotka się na wspólne sesje pracy, konsultując każdą wprowadzoną zmianę. Do wspomagania tego rodzaju pracy zespół korzysta z repozytorium GitLab, w którym dostępne są narzędzia code review.

Przewidywane funkcje do zademonstrowania w ramach odbioru częściowego

Zespół przewiduje, że do 10 stycznia zostaną wykonane i przetestowane następujące funkcjonalności:

- Implementacja serwera, który będzie w stanie połączyć się z klientami i prowadzić komunikację prywatną z innym klientem przy pomocy protokołu TCP,
- Implementacja obsługi komunikatów PASS, NICK, USER, QUIT, PRIVMSG, WHO, PING, PONG.

Funkcjonalności związane z tworzeniem, administrowaniem i zarządzaniem kanałami oraz komunikacją w obrębie kanałów stanowić będzie przedmiot dalszych prac projektowych w ramach etapu czwartego projektu (do 22 stycznia).

Harmonogram prac został wprzedzony przy okazji implementacji projektu końcowego. W ramach odbioru częściowego został przygotowany oddzielny plik ze zrzutami ekranu prezentującymi działanie serwera "demonstracja_działania.pdf".

Plan pracy z podziałem na tygodnie

- Tydzień 15-21 grudnia: Implementacja serwera, który będzie w stanie połączyć się z klientami i wysyłać/odbierać komunikaty przy pomocy protokołu TCP.
- Tydzień 22-28 grudnia: Przerwa.
- Tydzień 29 grudnia - 4 stycznia. Implementacja większości zaplanowanych w założeniach komend wraz z obsługą błędów.
- Tydzień 5-10 stycznia: Przeprowadzenie pełnego testowania aktualnej wersji rozwiązania (testy integracyjne). Przygotowanie odbioru częściowego.
- Tydzień 11-18 stycznia: Implementacja pozostałych niezrealizowanych wcześniej komend wraz z obsługą błędów oraz testowaniem.
- Tydzień 19-22 stycznia: Przygotowanie dokumentacji końcowej i prezentacja.

Harmonogram prac został wyprzedzony przy okazji implementacji projektu końcowego.

2. Opis najważniejszych rozwiązań funkcjonalnych

1. Architektura serwera to jednoprocesowy event-loop na epoll. Główna pętla serwera znajduje się w Server::run() (plik Server.cpp) i jest oparta o: gniazdo nasłuchujące TCP (socket/bind/listen), tryb nieblokujący (fcntl(..., O_NONBLOCK)), mechanizm epoll w trybie edge-triggered (EPOLLET). Model event-driven z epoll pozwala obsłużyć wielu klientów jednocześnie w jednym wątku, bez kosztów tworzenia i synchronizacji wielu wątków.

2. Przechowywanie i identyfikacja klientów - Klienci są przechowywani w Server::clients_ jako std::map<int, std::unique_ptr<Client>>, gdzie kluczem jest deskryptor socketu. Client (Client.hpp, Client.cpp) utrzymuje stan sesji klienta: nick_, username_, realname_, registered_, authenticated_, capNegotiation_,
bufor wejściowy buffer_ (sklejanie danych z TCP), lista kanałów użytkownika: std::set<std::string> channels_, metryki keep-alive: lastActivity_, lastPingSent_, awaitingPong_.

3. Parser wiadomości IRC i walidacja wejścia - Tokenizacja wiadomości realizowana jest przez MessageParser::parse() (MessageParser.cpp):
dzieli wiadomości po białych znakach, obsługuje parametr "trailing" (część po : traktowana jako jeden token). Walidacja nicków i kanałów: MessageParser::isValidNick() - sprawdza długość i dozwolone znaki w nicku, pierwszy znak musi być literą, ta funkcja jest również używana do walidacji username; MessageParser::isValidChannel() - sprawdza, czy początek nazwy kanału to # lub &, ma ona brak spacji i znaków sterujących oraz spełnia limit długości. Umożliwia to prostą tokenizację oraz walidację, które zabezpieczają serwer przed błędnymi komendami oraz ułatwiają implementację logiki komend.

4. "Command Handler" dla obsługi komend - CommandHandler (CommandHandler.hpp, CommandHandler.cpp) mapuje nazwę komendy na obiekt klasy pochodnej Command. Każda komenda ma własną klasę (*Command.cpp) i implementuje funkcję execute(), która odpowiada za rzeczywiste wykonanie komendy. CommandHandler::handleCommand() zapewnia odgórne zasady dla rozpatrzenia każdej komendy: wysłanie komendy bez uprzedniej rejestracji - wysłanie komunikatu 451 ERR_NOTREGISTERED (dla komend wymagających rejestracji), brak podanego uprzednio hasła komendą PASS skutkuje wysłaniem komunikatu 462 PASS required before registration (dla komend wymagających zalogowania hasłem), wysłanie nieznanej komendy skutkuje wysłaniem komunikatu 421 Unknown command. Rozdzielenie logiki komend na osobne klasy upraszcza rozwój projektu.

5. Model kanałów i uprawnień operatora - Channel (Channel.hpp, Channel.cpp): members_: std::set<int> socketów członków, operators_: std::set<int> socketów operatorów, topic_: temat kanału. ChannelManager (ChannelManager.hpp, ChannelManager.cpp) zapewnia: tworzenie/usuwanie kanału, dołączanie/opuszczanie, automatyczne usuwanie kanału, gdy ostatni użytkownik go opuści, usunięcie klienta z kanału z kontrolą operatora. Osobny ChannelManager centralizuje logikę kanałów i ułatwia ich zarządzanie.

6. Keep-alive PING/PONG i automatyczne rozłączanie - Serwer w Server::tickKeepAlive() cyklicznie: wysyła PING :<SERVER_NAME> co SERVER_PING_INTERVAL_MS czasu, oczekuje PONG (oznaczane przez Client::awaitingPong_), po przekroczeniu SERVER_PONG_TIMEOUT_MS rozłącza klienta z powodem "Ping timeout". Mechanizm pozwala wykrywać "martwe" połączenia i utrzymywać aktualny stan użytkowników/kanałów.

7. Logowanie zdarzeń - Inicjalizacja loggera: logger::init("app.log") (main.cpp, Logger.cpp). Logi trafiają równolegle: na konsolę i do pliku app.log, z wzorcem daty oraz czasu: [%Y-%m-%d %H:%M:%S]. Plik app.log jest resetowany przy ponownym uruchomieniu serwera. Logowanie do pliku jest kluczowe przy debugowaniu komunikacji sieciowej oraz przy tworzeniu testów integracyjnych.

3. Opis interfejsu użytkownika

Projekt nie posiada GUI, ponieważ implementuje serwer IRC, który z definicji działa stale w tle i odpowiada na wysłane do niego komendy. Po uruchumieniu serwera, można z niego skorzystać np. przy

pomocy takich narzędzi jak Weechat lub biblioteka libircclient. Serwer przetwarza poprawnie komunikaty wysyłane z użyciem takich narzędzi. Jedną z form interakcji z działaniem serwera można być obserwacja jego logów, które wypisywane są na terminal oraz zapisywane są w pliku app.log

Uruchomienie serwera

W celu ułatwienia uruchomienia projektu został on zdockeryzowany. Po sklonowaniu repozytorium projektu należy użyć poniższej komendy co poskutkuje uruchomieniem serwera na porcie 6667:

```
docker compose up
```

4. Postać plików konfiguracyjnych i logów

Pliki konfiguracyjne

- Config.hpp zawiera stałe liczbowe oraz domyślne parametry serwera - np. "constexpr int DEFAULT_PORT = 6667;".
- TestConf.cpp zawiera zmienne i klasy pomocnicze, które umożliwiają efektywne przeprowadzanie testów - m.in. symulacja klienta z wykorzystaniem biblioteki libircclient, zbieranie komunikatów serwera do weryfikacji.

Logi

Plik logu serwera: "app.log", format logów: [YYYY-mm-dd HH:MM:SS] [level] <wiadomość>. Przykładowe wpisy (fragment):

```
[2026-01-03 22:24:22] [info] IRC server started
[2026-01-03 22:24:32] [info] New client connected: 127.0.0.1:41844.
Socket: 6
[2026-01-03 22:24:33] [debug] [RECEIVED from 6] PASS 123456
[2026-01-03 22:24:33] [trace] PASS message received
```

5. Wykaz wykorzystanych narzędzi

Język i środowisko: C++, Linux (wykorzystanie m.in. epoll, fcntl, sygnały POSIX).

Budowanie projektu: make oraz g++ (zgodnie z Makefile).

W celu konteneryzaji projektu użyte zostało narzędzie Docker i Docker compose.

Biblioteki:
spdlog - logowanie (konsola + plik), GoogleTest (gtest) - framework testowy, libircclient - klient IRC używany w testach, boost - biblioteka do uproszczenia komunikacji sieciowej w testach.

Narzędzie pomocnicze użyte do prezentacji projektu: Weechat. Testy serwera są kompatybilne z rozszerzeniem VSCode "C++ TestMate".

W celu skontrolowania jakości rozwiązania zostało użyte narzędzie Valgrind, które zwróciło brak problemów:

```
[2026-01-05 01:16:55] [info] Shutting down server...
[2026-01-05 01:16:55] [info] Server has shut down
==57071==
==57071== HEAP SUMMARY:
==57071==     in use at exit: 0 bytes in 0 blocks
==57071==   total heap usage: 6,065 allocs, 6,065 frees, 502,902 bytes allocated
==57071==
==57071== All heap blocks were freed -- no leaks are possible
==57071==
==57071== For lists of detected and suppressed errors, rerun with: -s
==57071== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

6. Opis testów

Testy zawarte są w folderze /tests i są podzielone na pliki odpowiadające testom poszczególnych komend lub obsługiwanych błędów. Do konfiguracji części testów został użyty plik TestConf.cpp.

Testy mają charakter integracyjny i są uruchamiane w ramach gtest. Eksplorują one realne scenariusze połączenia klienta z serwerem, w którym symulowane są przypadki użycia różnych poprawnych i niepoprawnych komend.

Zastosowane testy przechodzą pozytywnie, co ukazuje poprawne działanie zaimplementowanych komend i komunikatów.

Sposób uruchomienia testów

W celu uruchomienia testów, należy użyć komendy `make test` a następnie za pomocą wywołania pliku bin/irc_tests rozpocząć testowanie. Zaimplementowane testy są kompatybilne z popularnym rozszerzeniem VSCode "C++ TestMate".