DOKUMENTACJA PROJEKTOWA

System współbieżnej edycji pliku tekstowego

Jan Prugarewicz - klient webowy
Jakub Świerczyński - serwer
Michał Makoś - klient desktopowy
Bartłomiej Wolski - klient administratorski

Spis treści:

1. Opis problemu

System służący do współpracy online nad edycją dokumentu tekstowego. Każdy użytkownik mający uprawnienia do edycji dokumentu powinien móc dowolnie manipulować jego treścią (usuwać i dodawać elementy). System zapewnia uwierzytelnienie - rejestrację oraz logowanie. Aplikacje posiadają proste interfejsy graficzne, a nawigować przy edycji można za pomocą myszki lub strzałek. Możliwa jest jednoczesne usuwanie wielu znaków jednocześnie.

2. Założenia

- Edycja dokumentu polega na dodawaniu i usuwaniu znaków
- Dwóch lub więcej użytkowników może edytować ten sam fragment tekstu
- Dostęp do dokumentu jest przyznawany poprzez wpisanie unikalnego kodu id dokumentu lub poprzez nadanie uprawniania użytkownikowi przez twórce dokumentu
- Nad jednym dokumentem nie może pracować więcej niż N użytkowników
- Jest wyznaczony maksymalny rozmiar jaki dokument może osiągnąć
- Można edytować tylko jeden dokument na raz

3. Wymagania funkcjonalne

- Możliwość wspólnej edycji dokumentu
- Możliwość zapisywania dokumentu na dysku lokalnym przez aplikacje kliencką
- Możliwość wyboru dokumentu do edycji
- Klient administratorski ma dostęp tylko do danych o użytkownikach i metadanych
- Administrator może tworzyć, edytować i usuwać konta innych użytkowników
- Bezpieczny kanał przesyłania informacji
- Odporność na zakłócenia/przerwania w ruchu sieciowym

4. Wymagania niefunkcjonalne

- Aplikacje powinny działać bez opóźnień, tak aby wszystkie zmiany dokonane w dokumencie były widoczne w interfejsach użytkowników w określonym czasie
- Powinien w sposób wydajny korzystać z zasobów sprzętowych, zarówno serwera jak i użytkownika
- Obsługa wielu użytkowników jednocześnie
- Zapewnienie pełnej architektury - serwer, aplikacje klienckie i aplikacja administracyjna
- Raportowanie błędów użytkownikowi w razie wystąpienia takiego

5. Główne przypadki użycia

5.1. Logowanie do aplikacji

Aktorzy: Użytkownik lub administrator
Zdarzenie inicjujące: Użytkownik uruchomił aplikacje i połączył się z serwerem
Scenariusz główny:

1. Użytkownik wprowadza login oraz hasło.
2. Użytkownik zatwierdza wprowadzone dane.
3. Serwer wysyła do klienta statement o możliwości zalogowania się.
4. Aplikacja wysyła do serwera wiadomość z loginem oraz hasłem.
5. Serwer sprawdza czy istnieje konto użytkownika powiązane z podanymi danymi logowania.
6. System potwierdza istnienie konta o podanych danych logowania i przesyła id sesji.
7. Użytkownik zostaje zalogowany.

Scenariusz alternatywny 2 - użytkownik wprowadził złe dane:

1-5. Jak w scenariuszu głównym.
6. Serwer odsyła klientowi Statement informujący go o niepoprawności przesłanych danych
7. Użytkownik zostaje powiadomiony o potrzebie ponownego logowania z powodu źle wprowadzonych danych.
8. Powrót do kroku 1.

5.2. Rejestracja nowego użytkownika

Aktorzy: administrator
Zdarzenie inicjujące: Administrator w panelu administracyjnym kliknął przycisk utwórz konto
Scenariusz główny:

1. Administrator wprowadza login.
2. Administrator wprowadza hasło.
3 Administrator wprowadza hasło powtórnie.
4. Administrator klika przycisk utwórz konto.
5. Aplikacja sprawdza poprawność danych.
6. Dane poprawne aplikacja wysyła wiadomość do serwera z poleceniem stworzenia konta.

Scenariusz alternatywny 2 - Wprowadzone hasła nie zgadzają się:

1-5. Jak w scenariuszu głównym.
6. Aplikacja wyświetla monit o niezgadzających się hasłach
7. Powrót do kroku 3.

5.3. Wybór pliku do edycji

Aktorzy: klient, serwer
Zdarzenie inicjujące: klient wybrał opcję: Open from server
Scenariusz główny:

1. Aplikacja wysyła do serwera statement typu 7, czyli FILES_REQUEST.
2. Serwer wysyła listę wszystkich plików dostępnych do edycji.
3. Klient wybiera nazwę pliku, który będzie chciał edytować.
4. Aplikacja wysyła do serwera wiadomość z nazwą pliku do edycji.
5. Serwer sprawdza czy plik jest w trakcie edycji.
6. Serwer wysyła bufory znaków do klienta.
7. Aplikacja ładuje bufory znaków i wyświetla w oknie do edycji.

Scenariusz alternatywny 2 - Plik nie jest aktualnie w trakcie edycji

1-5. Jak w scenariuszu głównym.
6. Serwer ładuje plik i konwertuje na bufory znaków, ładując je jednocześnie do pamięci podręcznej.
7. Powrót do kroku 6.

5.4. Edycja dokumentu

Aktorzy: kilka aplikacji, serwer
Zdarzenie inicjujące: klient wprowadził zmiany do edytora oraz minął określony czas czekania.
Scenariusz główny:

1. Jedna z aplikacji tworzy i wysyła wiadomość typu EDIT do serwera z zawartością buforów zmian.
2. Serwer rozdziela zmiany na bufory znaków dodanych oraz usuniętych i dopisuje je do buforów w pamięci podręcznej.
3. Serwer przesyła zmiany do pozostałych aplikacji.
4. Pozostałe aplikacje klienckie odbierają wiadomość ze zmianami
5. Aplikacje updateują tekst w edytorze dodając do niego znaki z bufora znaków dodanych i usuwając znaki z bufora znaków usuniętych.

6. Diagramy sekwencji

Diagram główny - połaczenie, edycja i koniec

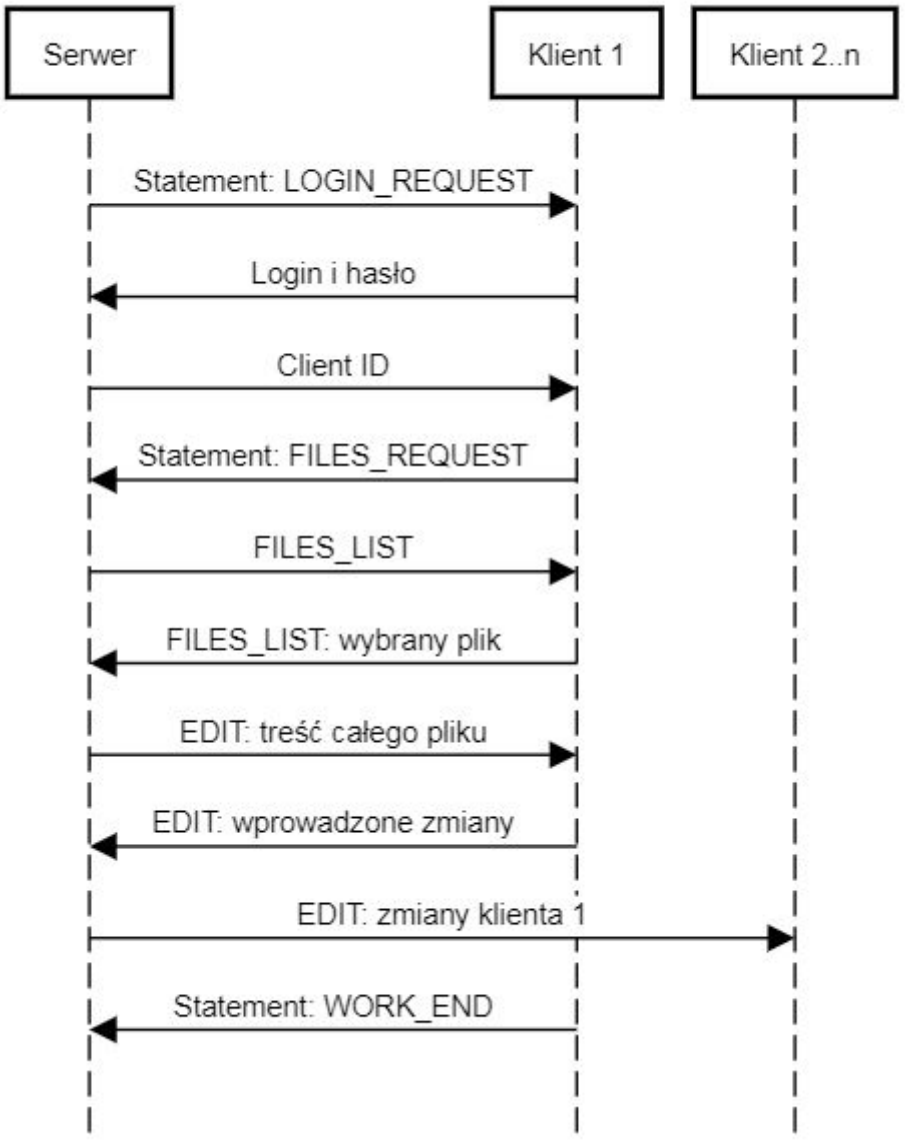
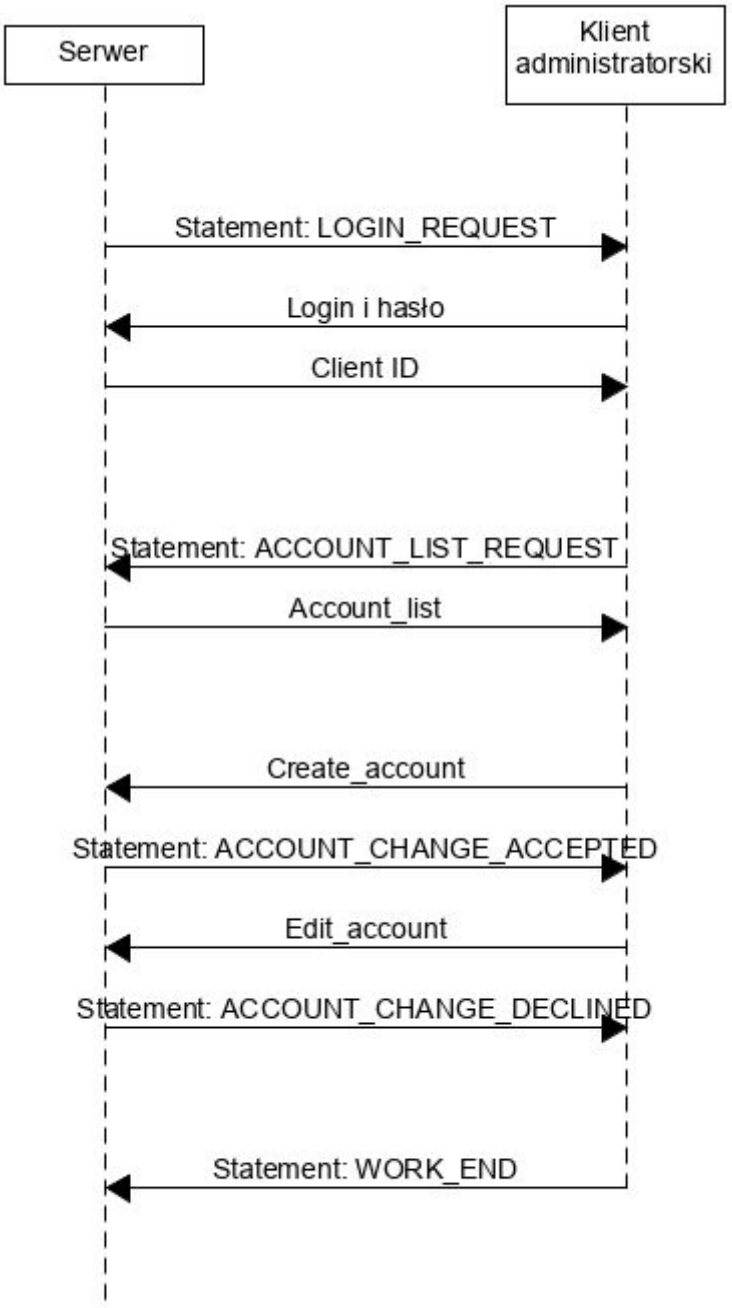


Diagram pomocniczy/administratorski - połączenia, praca, koniec



7. Technologie i platforma docelowa

Do stworzenia serwera został wykorzystany język C++, do klienta - Java, zaś klient administratorski jest napisany w Pythonie.
Platformą docelową projektu jest Linux.

Aplikacja kliencka w Javie działa na Windowsie,

8. Protokół

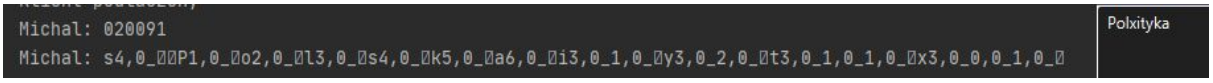
Ponieważ wiadomości są zmiennej długości, a odbierane są przy pomocy funkcji, które przyjmują jako argument liczbę znaków do odebrania, to musimy najpierw wysłać header o stałej długości, który zawiera informacje o przesyłanej po nim wiadomości.

8.1. Header

Header ma za zadanie poinformować o typie przychodzącej wiadomości oraz o jej długości. Ma ustaloną postać 6-bajtowego stringa, który wygląda następująco:

- Pierwsze 2 bajty informują o typie wiadomości
- Kolejne cztery bajty informują o długości wiadomości.

Przy czym powyższe informacje, które są liczbami są wysyłane jako int sparsowany na stringa. Czyli np. dla wiadomości typu 2 i długości 27 header wygląda następująco: “020027”. Możemy zaobserwować to w działaniu aplikacji na przykładzie wysyłanych zmian, opisanych w algorytmie synchronizacji:



8.2. Typ wiadomości

Do obsługi komunikacji między serwerem i aplikacjami wykorzystujemy następujące typy wiadomości:

- LOGIN = 0 - jest to typ wiadomości służący do przesyłania danych do logowania w formacie: “login\nhasło”.
- STATEMENT = 1 - typ wiadomości służący do przekazywania kodów informacji. Szczegółowy opis w sekcji Statement.
- EDIT = 2 - typ wiadomości służący do przekazywania zmian w plikach, a także plików dopiero otworzonych. Dokładny opis znajduje się w sekcji algorytmu synchronizacji.
- CLIENT_ID = 4 - typ wiadomości do przekazania ID klienta, które potrzebne jest do prawidłowego działania synchronizacji tekstu.
- FILES = 5 - typ wiadomości, do przesyłania nazw plików dostępnych do edycji, a także nazwy pliku do edycji wybranego przez aplikację kliencką.
- CREATE_ACCOUNT = 6 - typ wiadomości służący to przesłania danych logowania nowego użytkownika postaci “login\nhasło”.
- EDIT_ACCOUNT = 7 - typ wiadomości pozwalający zmienić ustawiania już istniejącego konta postaci “oldlogin\nlogin\nhasło”, gdzie jeśli nie zmieniamy któregoś z parametrów login lub hasło jest on oznaczony “*”

- DELETE_ACCOUNT = 8 - typ wiadomości służący do usunięcia konta ,przesyłamy nazwę konta, które chcemy usunąć
- ACCOUNT_LIST = 9 - typ wiadomości służący do przesłania listy istniejących kont wraz z informacją o statusie online/offline

8.3. Statement

Do przekazywania komunikatów służy typ wiadomości STATEMENT, który jest reprezentowany jako integer określający komunikat. I tak:

- KEEP_ALIVE = 0 - statement wysyłany do serwera przez aplikację, aby upewnić się, że aplikacja nadal jest podłączona do serwera. Również ten typ statementu jest zwracany przez serwer w celu potwierdzenia połączenia.
- REQUEST_LOGIN = 1 - statement wysyłany przez serwer przy rozpoczęciu sesji, aby powiadomić użytkownika, że ten może już się zalogować.
- LOGIN-ACCEPTED = 2 - statement służący do potwierdzenia przez serwer poprawnego zalogowania się.
- LOGIN_REJECTED = 3 - statement służący do poinformowania przez serwer o niepoprawnym zalogowaniu się.
- LOG_OUT = 4 - statement służący do powiadomienia klienta, że ten został wylogowany z serwera, a także aby klient powiadomił serwer o chęci wylogowania się.
- WORK_END = 5 - statement służący do powiadomienia serwera o chęci zakończenia pracy, a także powiadomienia klientów przez serwer o kończeniu pracy serwera.
- FILE_LIST_REQUEST = 7 - statement służący do wysyłania żądania przez aplikację o wysłanie listy plików dostępnych do edycji.
- ACCOUNT_LIST_REQUEST = 8 - statement służący do wysyłania żądania przez aplikację administrującą o wysłanie listy zarejestrowanych kont.
- ACCOUNT_CHANGE_ACCEPTED = 9 - statement służący do potwierdzenia przez serwer poprawnej edycji konta (w tym usunięcia lub utworzenia).
- ACCOUNT_CHANGE_DECLINED = 10 - statement służący do poinformowania przez serwer o odrzuceniu edycji konta (w tym usunięcia lub utworzenia).

9. Aplikacja kliencka

9.1. Algorytm synchronizacji

Aby dokument mógł być poprawnie zsynchronizowany należy reprezentować dane w postaci danych CRDT (Conflict-free replicated data), czyli takich, dla których operacje dodawania i usuwania są rozłączne i przemienne.

Możemy przyjąć strategię wstawiania znaków po określonych znakach np. gdy “Wind” chcemy zamienić na “Window” to “ow” chcemy wstawić po słowie “Wind”. Wiadomo jednak, że może istnieć wiele słów “Wind” w tekście, więc nie jest to dobre rozwiązanie, w dodatku mocno złożone obliczeniowo.
Możemy również przyjąć strategie wstawiania tekstu na określonej pozycji. Rozpatrzmy przypadek, gdzie w tekście “Ala ma kota” użytkownik 1 chce uzyskać tekst “Ala ma złego kota”, a w tym samym czasie użytkownik 2 chce uzyskać tekst “Ala nie ma kota”. Wtedy kiedy najpierw przetworzymy żądanie użytkownika 1 to uzyskamy satysfakcjonujący wynik “Ala nie ma złego kota”. Jeśli natomiast przetworzymy operacje w odwrotnej kolejności uzyskamy wynik “Ala niezłego ma kota”, co jest niby fajne, ale chyba nie o to nam chodziło - operacje nie są przemienne bo nie dają tego samego wyniku przy przetwarzaniu w innych kolejnościach. Aby tak było, każdy znak musi posiadać swoje unikalne ID, tak aby każdy znak był rozróżnialny. Dodatkowo to ID powinno zapewniać prawidłowe sortowanie znaków. Najprościej jest to sobie wyobrazić w ten sposób, że mamy słowo “Barrakuda” i każdemu znakowi w tym słowie przyporządkowujemy ID zapewniające posortowanie:
B a r a k u d a
1 2 3 5 6 7 8 9
Teraz jeśli zauważymy literówkę, bo to słowo pisze się przez dwa ‘r’, to możemy wstawić tam ten znak przyporządkowując mu ID spomiędzy liczb 3 i 4 a więc np. 3,5.
B a r r a k u d a
1 2 3 3,5 4 5 6 7 8
Numeracja celowo zaczyna się od liczby 1, na wypadek, gdy będziemy chcieli wstawić znak na pozycji 0. Wtedy znak ten będzie miał ID = 0,5. Można więc przyjąć (i tak też reprezentuję dane w aplikacji), że na początku tekstu stoi zawsze znak kontrolny, który nie ma żadnej znaczącej wartości, nie należy do dokumentu, a jego ID wynosi 0. W takiej reprezentacji może pojawić się problem skończonej reprezentacji oraz niedokładności reprezentacji liczb zmiennoprzecinkowych.
Dlatego też rzeczywista reprezentacja tych ID nie składa się z 1 liczby zmiennoprzecinkowej, a z wektora liczb typu Integer, gdzie każda następna wartość reprezentuje jakby kolejną liczbę po przecinku.
Jednak występuje jeszcze drugi problem, co jeśli użytkownicy wstawią ten sam znak w to samo miejsce, albo co gorsza inny znak w to samo miejsce. Wtedy będą miały one to samo ID. Aby tego uniknąć, to każdej liczby z ID znaku dopiszemy unikalne ID klienta, który utworzył dany znak i które jest przydzielone przez serwer.

Tak więc ostatecznie w naszym projekcie taka reprezentacja wygląda tak, że każdemu znakowi przyporządkowany jest wektor par, który reprezentuje pozycję danego znaku, czyli jego ID. Para ta przechowuje liczbę naturalną, która odpowiada za podstawową pozycję znaku w tekście oraz liczbę naturalną reprezentującą id użytkownika który utworzył ten znak.
Przyjrzyjmy się działaniu takiego algorytmu. Najpierw użytkownik 1 wpisuje słowo “Polska”:

P	o	l	s	k	a
[1,1]	[2,1]	[3,1]	[4,1]	[5,1]	[6,1]

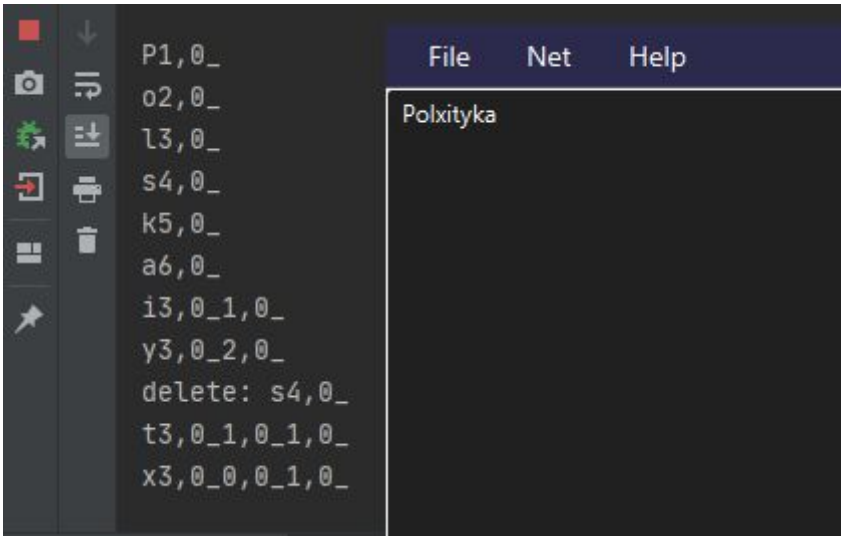
Teraz użytkownik 2 wstawia literę ‘i’ po literze ‘l’ a potem wstawia ‘y’ i usuwa ‘s’:

P	o	l	i	y	k	a
[1,1]	[2,1]	[3,1]	[3,1][1,2]	[3,1][2,2]	[5,1]	[6,1]

Teraz użytkownik 1 wstawia ‘t’ i otrzymujemy:

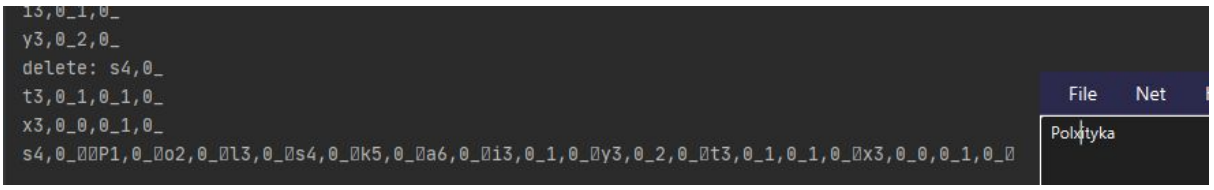
P	o	l	i	t	y	k	a
[1,1]	[2,1]	[3,1]	[3,1][1,2]	[3,1][1,2][1,1]	[3,1][2,2]	[5,1]	[6,1]

Co jednak jeśli chcemy wstawić literę przed ‘i’? Nie możemy przecież rozszerzyć ID ‘l’: [3,1] bo już to zrobiliśmy i dostalibyśmy ID = ‘i’. Dlatego właśnie numerujemy id od 0. Możemy teraz wziąć ID ‘i’ usunąć ostatni element, dodać liczbę 0 i na końcu dodać liczbę w odpowiedniej kolejności: [3,1][0,1][1,1].
Teraz, aby uzyskać poprawną kolejność znaków, wystarczy posortować po pierwszych elementach id w parach, a w przypadkach, kiedy są one równe, wystarczy porównać ID użytkownika, który znak dodał.
Jak to działa w praktyce?



Dodawałem znaki kolejno tak, jak opisałem w przykładzie. Możemy zobaczyć, że dostaliśmy spodziewany output (oprócz ID, bo miałem ustawione na 0).

Na powyższym screenie możemy także zobaczyć, jak wygląda reprezentacja liczb przy wysyłaniu do serwera - musi to byc string. Pierwszy znak jest zawsze znakiem, który chcemy wstawić (lub usunąć). Następnie pary reprezentujące pozycję są oddzielane podłogą - ‘_’, a liczby w parach oddzielone są przecinkiem - ‘,’. Ponieważ buforujemy zmiany i wysyłamy kilka naraz, to kolejne znaki oddzielone są znakiem z ASCII o kodzie 29 czyli separator grupy, natomiast bufor znaków usuniętych od bufora znaków dodanych oddzielony jest znakiem o kodzie 28 w ASCII czyli separatorem pliku. Tak więc finalna wiadomość wysłana na serwer po powyższych operacjach wygląda następująco:



gdzie przekreślone kwadraciki są oczywiście separatorami. Jak widać najpierw został wysłany usunięty znak s, następnie separator znaku, potem separator buforów i następnie znaki dodane oddzielone separatorami.

Gdy taki ciąg jest przekazany do klienta, ten dla każdego znaku szuka miejsca gdzie go wstawić a następnie wstawia go na to miejsce.
Zmiany są także zapisywane na serwerze, jednak bez stałego sortowania, gdyż jest to niepotrzebne.
Aby zmniejszyć złożoność pamięciową algorytmu serwer po zakończeniu pracy z plikiem, aby go zapisać, usuwa wszystkie znaki znajdujące się w wektorze znaków usuniętych z wektora znaków dodanych. Następnie sortuje wektor znaków dodanych i zapisuje do pliku tylko znaki (w odpowiedniej kolejności). Następnie przy wznowieniu pracy nad plikiem, serwer na nowo przydziela znakom ID, które będzie dla każdego znaku wektorem o długości 1. Dzięki temu, przy każdym restarcie pliku, jego objętość jest minimalizowana.

9.2. Architektura

Program jest podzielony na pliki w następujący sposób:

- Connect.java - Klasa odpowiadająca za obsłużenie okna z danymi do logowania oraz połączenia. Po wprowadzeniu tych danych i kliknięciu “Log in” klasa tworzy nowego klienta i wywołuje jego metody w celu połączenia się.
- Client.java - klasa obsługująca ruch sieciowy. To w niej tworzone są wątki programu oraz notatnik. Odpowiedzialna jest za połączenie się z serwerem, odbieranie i przetwarzanie przychodzących wiadomości od serwera oraz przetwarzanie i wysyłanie żądań od notatnika do serwera.
- Message.java - klasa pomocnicza dla klasy Client. Opakowuje ona wszystkie typy wiadomości. Dzięki niej można utworzyć wiadomość jako obiekt tej klasy odpowiednim konstruktorem a następnie odpowiednią metodą otrzymać gotową wiadomość do wysłania lub otrzymać poszczególne części otrzymanej wiadomości.
- Notepad.java - klasa notatnika. Odpowiada ona przede wszystkim za obsługę notatnika graficznego, a także przetwarza wprowadzone do niej dane stosując algorytmy synchronizacji.
- Encryption.java - klasa odpowiadająca za szyfrowanie. Należy ustawić konstruktorem odpowiedni klucz a następnie każdą wiadomość szyfrować oraz rozszyfrowywać odpowiednimi metodami tej klasy.
- Interfejs graficzny jest zapisany w plikach .fxml, do których należą także pliki .css oraz obrazy definiujące style okien.

Program po poprawnym połączeniu z serwerem oraz zalogowaniu jest dzielony na 4 wątki:

- wątek główny programu, czyli interfejs graficzny, który obsługuje notatnik, w razie potrzeby wysyła wiadomości do serwera np. z prośbą o listę plików do edycji
- wątek wysyłający wiadomości do serwera z buforami znaków dodanych i usuniętych, regularnie co określoną liczbę sekund, ale tylko gdy przynajmniej jeden z buforów jest niepusty
- wątek czytający, czyli wątek, który na bieżąco odbiera wiadomości od serwera i obsługuje żądania - głównie wiadomości z buforami znaków od innych użytkowników
- wątek keep-aliver, czyli wątek, który co określoną liczbę sekund sprawdza czy jest połączenie z serwerem, wysyłając do niego wiadomość typu statement z

statementem = 0, czyli keep-alive. Następnie czeka określoną liczbę sekund na wiadomośc zwrotną. Jeśli jej nie otrzyma, to wyświetla monit o zerwaniu połączenia. Po tym monicie użytkownik może zrestartować połączenie. Ten request jest wysyłany tylko wtedy gdy przez czas, kiedy keep-alive śpi nie przyjdzie żadna wiadomośc od serwera, czyli flaga isAlive = false. Każda wiadomość od serwera podnosi tą flagę do wartości true.

9.3. Połączenie

Po uruchomieniu aplikacji wyświetla się okno z polami tekstowymi. Należy podać ip serwera, port, login i hasło. Można użyć defaultowych opcji połączenia poprzez zaznaczenie pola “set default”. Po naciśnięciu przycisku “Login” aplikacja próbuje się połączyć zgodnie z diagramem sekwencji połączenia:

- Najpierw łączy się na socket. W przypadku niepowodzenia wyświetla monit o niepowodzeniu, po którym można próbować jeszcze raz. W przypadku powodzenia ustawia flagę ‘connected’ na true.
- Jeśli flaga ‘connected’ jest true aplikacja próbuje zalogować się do serwera. W przypadku niepowodzenia, można spróbować zalogować się jeszcze raz (już nie trzeba się łączyć na nowo). W przypadku powodzenia otwierana jest aplikacja edytora tekstu.

9.4. Niepożądane rozłączenie

Dzięki wątkowi keep-alive możemy wykrywać sytuację rozłączenia się z serwerem. Gdy taka sytuacja nastąpi, flaga isAlive zostaje opuszczona i wątki zawieszają komunikację z serwerem. Zostaje wyświetlony monit o przerwaniu połączenia z serwerem. Tekst można dalej edytować, lecz zmiany zostaną wysłane na serwer dopiero po ponownym połączeniu się. W tym celu należy z panelu na górze wybrać opcję net->reconnect. Wtedy aplikacja spróbuje na nowo się połączyć z serwerem, oraz w przypadku powodzenia zalogować, używając danych do logowania zapisanych w pamięci podręcznej. Jeśli któraś z tych operacji się nie powiedzie, na ekranie pojawia się odpowiedni monit z informacją. Można spróbować nawiązać połączenie jeszcze raz lub zrestartować aplikację. Przedtem można zapisać plik lokalnie na swoim komputerze za pomocą opcji file->save.

9.5. Kończenie pracy

Aplikację można zakończyć przyciskając X lub wybierając opcję file->quit. Obie opcje dają ten sam rezultat. Najpierw wysyłane są bufory zmian, aby mieć pewność, że ostatnie zmiany zostaną zapisane na serwerze. Następnie wysyłany jest do serwera statement WORK_END o kodzie 5. Następnie flaga isRunning jest opuszczana co powoduje zakończenie pracy wszystkich wątków. Na końcu zamykany jest wątek główny programu.

9.6. Pozyskanie pliku do edycji

Po poprawnym połączeniu się z serwerem ukazuje się okno do edycji pliku. Jednak wszystko co tam wpiszemy jest nieważne dopóki nie wybierzemy pliku do edycji. W tym celu należy wybrać opcję file-open from server. Wtedy aplikacja wyśle do serwera żądanie o listę dostępnych plików, czyli statement FILE_REQUEST o kodzie 7. Następnie po odebraniu listy plików od serwera (wiadomośc typu FILES, nazwy rozdzielone znakiem '\n') wyświetlana jest lista plików do wyboru. Należy wybrać plik poprzez kliknięcie na nazwę wybranego pliku i zamknięcie okna. Wtedy do serwera zostanie wysłana wiadomość typu FILES z nazwa wybranego pliku. Po otrzymaniu buforu znaków dodanych od serwera, tworzony jest lokalnie nowy plik, który jest następnie wyświetlany w oknie do edycji.

10. Aplikacja administratorska

10.1 Architektura

Na architekturę klienta administratorskiego składają się trzy główne komponenty:

- socket_client.py - jest to element obsługujący ruch sieciowy. Odbiera dane od serwera i przekazuje je kontrolerowi, a także wysyła do serwera wiadomości przygotowane przez kontroler.Uruchamiany jako wątek.
- client_controller.py - moduł o roli kontrolera, decydującego co robić z informacjami dostarczanymi od interfejsu graficznego i klienta socketowego.Uruchamiany jako wątek.
- GUI.py - Część odpowiedzialna za dostarczenie interfejsu graficznego.Odbiera dane od użytkownika i informuje kontroler poprzez wrzucanie danych do system_messages kontrolera oraz przekazuje dane(np:logowania) do system_variables.Uruchamiany w głównym procesie.

moduły użytkowe:

- encryptor.py - dostarcza funkcje szyfrujące.
- my_errors.py - klasy błędów pomocna przy obsłudze wyjątków
- protocol_wrapper - dostarcza funkcje opakowujące dane w protokoły
- locked_resources - zasoby przechowujące dane synchronizowane
- unit_tests - testy jednostkowe podstawowych elementów systemu

10.2 Połączenie

Po uruchomieniu, aplikacja próbuje połączyć się z serwerem na adres wskazany przez plik settings.txt. Jeśli serwer jeszcze nie jest uruchomiony(czyli nie działa) aplikacja wyświetli monit o straconym połączeniu z serwerem i przycisk "reconnect", pozwalający na ponowną próbę połączenia z serwerem. Jeśli połączenie zostanie nawiązane,wątek Socket_client

wystartuje i zostanie wyświetlony monit oczekiwania na login request serwera oznaczający rozpoczęcie komunikacji.

10.3 Niepożądane rozłączenie

W przypadku nagłego zamknięcia serwera (odebrania “pustej wiadomości“= b””) aplikacja kończy wątek klienta socketowego i wyświetla informacje o straconym połączeniu z serwerem oraz przycisk “reconnect”, którego wciśnięcie powoduje ponowną próbę połączenia z serwerem. Brak aktywność w komunikacji powoduje wysłania wiadomości KEEP_ALIVE i jeśli dalej nie otrzymujemy wiadomości od serwera aplikacja kończy wątek i informuje o braku połączenia.

10.4 Kończenie pracy

Aby zakończyć prace należy wcisnąć przycisk “x”. Wszystkie wątki zostaną zakończone poza socket_client, który prześle wiadomość WORK_END i też zakończy pracę.

11. Serwer

11.1 Architektura

Aplikacja serwera podzielona jest na następujące moduły:

- Moduł główny serwera (pliki Server.h i Server.cpp) spinający wszystkie moduły podrzędne i realizujący główną pętlę programu, oczekiwanie na dostępne gniazda, wysyłanie/odbieranie wiadomości, tworzenie zbiorów odbierających i wysyłających gniazd, przyjmowanie nowych połączeń, obsługa zakończonych połączeń.
- Moduł nadzorujący stan klientów (ClientsMonitor.h ClientsMonitor.cpp) zawierający informacje o stanie wysyłania/odbierania na danym gnieździe oraz ilości bajtów do wysłania/odebrania przydatnych w obsłudze niepełnych odebrań/wysłań. Moduł ten też nadzoruje zbiór klientów zalogowanych i odrzuconych przy logowaniu.
- Moduł obsługujący bufory odbierające (RecvBuffers.h/.cpp) zastosowany do odtworzenia całej wiadomości przy częściowych odbiorach.
- Moduł obsługujący logowanie (LoginHandler.h/.cpp), który odtwarza otrzymane dane logowania, porównuje z danymi w pliku z istniejącymi użytkownikami w systemie oraz dodaje nowych użytkowników do listy.
- Moduł obsługujący operacje sieciowe (Networking.h/.cpp) docelowo miał obsługiwać wszystkie operacje związane z wysyłaniem i odbieraniem wiadomości oraz oddelegowywanie ich obsługi do odpowiednich podmodułów. W obecnej wersji moduł ten posiada kolejki wiadomości do wysłania w celach zachowania synchroniczności np. otrzymywanie przez serwer dużej ilości wiadomości typu *EDIT*. Oprócz tego monitoruje typ wiadomości, która ma zostać wysłana jako następna lub serwer jest w trakcie jej wysyłania.

- Moduł wspomagający obsługę współdzielonego notatnika (NotepadHandler.h/.cpp), który opakowuje podstawowe API z pliku Note.cpp, oprócz tego nadzoruje stan klientów edytujących dokument, przechowuje zmiany wysłane przez klientów z wiadomości typu *EDIT*, czuwa nad poprawną kolejnością zmian zawartości pliku oraz monitoruje nowych klientów, którzy muszą pobrać zawartość pliku.
- Moduł nadzorujący pliki tekstowe (TextFileHandler.h/.cpp) zawierający informację o wybranych plikach tekstowych tzn. każdy moduł może posiadać własną instancję tej klasy jak np *LoginHandler*, który przechowuje informacje o pliku z danymi użytkowników. Docelowo użyteczny w celu dostarczenia informacji użytkownikowi o jego plikach, które może edytować.
- Moduł pomocniczy do manipulowania plikiem tekstowym (FileIOHandler.h/.cpp) używanym do otwierania i zamykania pliku, tworzenia nowego oraz zapisywania zawartości z buforów programowych lub też ekstrakcji danych z pliku do bufora.

Abstrakcja wiadomości:

Ze względu na naturę protokołu TCP bazującym na strumieniu bajtów postanowiono stworzyć klasę abstrakcyjną *Message* w celu sformalizowania komunikacji, po której dziedziczą konkretne typy wiadomości (np. *Header*, *Edit* itd.). Zachowanie klas dziedziczących bazuje na funkcjach zwracających typ wiadomości i reprezentację znakową danej wiadomości (serializacja).

11.2 Połączenie

Serwer oczekuje na gotowość gniazda nasłuchującego do odbioru wiadomości, po czym wysyłany zostaje *LOGIN_REQ* w celu powiadomienia klienta o obowiązku logowania do systemu, po otrzymaniu danych zostają one zweryfikowane i odpowiednio przy powodzeniu tzn. dane prawidłowe lub nowy użytkownik, wysyłany jest do klienta jego identyfikator (numer gniazda), zaś przy niepowodzeniu logowania, klientowi wysłany zostaje *LOGIN_REJECTED*.

11.3 Edycja

Zalogowany klient ma możliwość edycji pliku online, jeśli jest on pierwszym w systemie chcącym edytować to tworzony jest nowy plik. Każdy kolejny klient dołączający do edycji otrzymuje całość pliku obecnie edytowanego. Jeśli edytujący klient jest już zapisany jako aktywny w danym dokumencie to jego wiadomość ze zmianami zostaje wysłana reszcie klientów, a bufor programowy odpowiednio zmodyfikowany przy użyciu API notatnika. Przy kończeniu połączeniu przez klienta sprawdzana zostaje ilość aktywnych uczestników edycji, jeśli jest on ostatni to zawartość bufora zostaje zapisana do pliku o nazwie “id_<user_id>_<ilość plików tekstowych w folderze + 1>” w folderze *texts.*

11.4 Synchronizacja zmian

W celu zapewnienia synchroniczności zmian tekstu moduł wspomagający użycie notatnika nadzoruje kolejność zmian wysyłanych każdemu klientowi poprzez mapowanie deskryptora

gniazda na listę zmian do wysłania (struktura *PendingChanges* zawierająca informacje o nadawcy i długości wiadomości, w celu pobrania odpowiedniej długości znaków z bufora zmian). Taka struktura pozwala na stosunkowo łatwe nadzorowanie wysyłanych zmian, każdorazowo definiowana jest jedna tura wysłań, jedna tura zawiera n - 1 wysłań gdzie n to ilość obecnie edytujących użytkowników. Przed pobraniem zmian z bufora porównywane są: pierwsza struktura w liście z ostatnią zastosowaną zmianą (w postaci struktury *PendingChanges*) i ilość pozostałych wysłań w danej turze z zerem. Jeśli wysłano wszystko z danej tury to modelem do porównania jako ostatnia zmiana zostaje następna wiadomość z listy (poprzednia oczywiście zostaje usunięta po wysłaniu danych z nią powiązanych).