Architektura:

W naszej aplikacji używamy pythonowego backendu z plikiem serwera i klienta.

Nasza aplikacja pracuje w trybie przesyłania zakodowanych wiadomości wiadomości przez serwer do klientów przez funkcje send i receive w skryptach client.py. W nich kodujemy i dekodujemy wiadomości, serwer dostaje zakodowane ciągi znaków.

Prócz tego mamy dla pliki OTP i OTP\_binary. Oba zawierają klucze do szyfrowania wiadomości, jeden w intami z zakresu 0-256 jak kodowanie utf8, drugie znaki binarne. Określiliśmy z góry limit wiadomości do 2000 znaków, stąd generowane klucze mają z góry określoną długość, w OTP jest to 2000 liczb, w OTP\_binary 16 000 bitów.

Klucze można wygenerować skyptem generate\_OTP.py.

Prócz tego mamy dwa skrypty z funkcjami do szyfrowania i deszyfracji stringów, jeden z zwiększający modulo(256) zdekodowane znaki utf8 do int, a drugi kodujący na poziomie bitowym. Funkcje deszyfrujące przeprowadzają odwrotne operacje z tego samego klucza.

Jak widać, zrobiliśmy funkcjonalną aplikację zakładając każdorazowe dostarczenie gotowego klucza na potrzeby przekazania informacji. W obecnej wersji, jest on generowany raz i używany przez cały czas życia aplikacji. Implementację pod szyfrowanie kwantowe i menedżer takiego klucza omówimy później.

Lifecycle:

Całość zaczyna się od uruchomienia skryptu server.py i tam zaimplementowaliśmy generowanie klucza OTP (z zakresu 0-256 dla kodowania utf8) i jego binarnej wersji dla niskopoziomowego kodowania (operacji bitowych).

Później uruchamiamy client.py, w konsoli mamy zapytanie o hosta i wpisujemy 127.0.0.1. Dalej serwer bierze pierwszą wysłaną wiadomośc za nazwę clienta, tą wiadomość wysyłamy zaszyfrowaną i wyjątkowo deszyfrujemy na serwerze. Zrobiliśmy dwie bezpieczne wersje aplikacji nie dające dostępu serwerowi do szyfrowania:

-pierwsza szyfruje i serwer zapisuje nazwę użytkownika jako bezsensowny, ale unikalny zaszyfrowany ciąg znaków i tak go cały czas określa

-aplikacja bez nazw użytkowników, gdyby była używana jedynie między dwoma osobami.

Zostaliśmy jednak przy tej wersji, gdyż jest user friendly, a konieczność implementacji we właściwy sposób osobnej funkcji wysłania nazwy użytkownika wymaga czasu i większej wiedzy na temat bibliotek, z których dopiero co zaczęliśmy korzystać.

Dalej odbywa się czat, serwer oczekuje na wiadomości od klientów i rozsyła zakodowane ciągi do reszty użytkowników. Kolejni klienci łączą się wpisując jako adres 127.0.0.2 itd.

(w pycharmie na macu w niewiadomych przyczyn każdy musi wpisywać 127.0.0.1)

Podstawą wymiany informacji są funkcje send i receive używające funkcji szyfrujących i deszyfrujących z OTP\_binary\_chiffre.py. Można zmienić na OTP\_chiffre i zmienną key\_filename na “OTP” aby wrócić do wysokopoziomowego szyfrowania znaków.

Aplikacja rozłącza się wpisując “quit” w okno czatu.

Implementacja kwantowa:

Nasza aplikacja jest dla wielu użytkowników naraz. Jednak dla uproszczenia zakładamy, że nasza aplikacja jest tylko dla dwóch osób.

Chcemy zrobić szyfrowanie wiadomości za każdym razem innym kluczem, a więc każde wysłanie wiadomości to też generowanie klucza.

Opiszemy implementację generatora klucza pod algorytm BB84. Zakładamy, że Alice i Bob, mają swoje komputery które mają kanał wymiany kwantowy i klasyczny, poza naszym kanałem wiadomości. U nas w projekcie będą to po prostu foldery pierwszego i drugiego klienta, które będą dostępne dla obu stron do zapisu plików.

Jedno przesłanie się wiadomości to uruchomienie funkcji send u jednego klienta i jedno wykonanie się funkcji receive u drugiego. Zróbmy tak, żeby za każdym razem oboje mieli plik “Abase1” (bazy Alice, podstawowa), “Bbase1” (bazy Boba, do odczytu), oraz “key1”.

Przy uruchamianiu się funkcji send porównujemy bazy i generujemy plik “base” złożony z 0 i 1 do odrzucania, albo akceptowania bitu z klucza. Dalej za pomocą tego pliku redukujemy klucz z “key1”, do pliku “key” i za jego pomocą szyfrujemy wiadomość. Dalej generujemy klucz i bazy na kolejną wiadomość do plików “Abase1” i “key1”, nadpisując stary, użyty klucz i bazy. Ponadto zapisujemy je do plików, tym razem “Abase2” i “key2” (nie chcemy nadpisać starego, jeszcze nie użytego przez klienta 2 klucza), do folderu drugiego klienta. Zauważmy, że dotychczas, wszystko zapisywaliśmy tylko u nas. Teraz “transportujemy” informację o bazach i kluczu do folderu drugiego klienta. W rzeczywistości, przesłalibyśmy mu spolaryzowane fotony kanałem kwantowym i informacje o bazach zwykłym kanałem. U nas klucz to może być ciąg 1,2,3,4 oznaczający:

* 1,2 - |0>, |1> w pierwszej bazie
* 3,4 - |0>, |1> w drugiej bazie.   
  dekodowanie odbywałoby się braniem swoich baz i weryfikacji, jeżeli na danym miejscu jest np. 3, a my mamy 0 (bo wybraliśmy bitowy zapis baz), czyli pierwszą bazę, losujemy stan. Jest to praktycznie bez sensu, ale pamiętajmy, że w rzeczywistości byłyby to fotony mnierzone polaryzajcą, nie byłoby tego problemu.  
  Dopiero na koniec wysyłamy zaszyfrowaną wiadomość, aby funkcja receive u klienta 2 uruchomiła się po zakończeniu przesyłania baz i klucza.

Funkcja receive uruchamia się u klienta dysponując plikami “Abase1”, “Bbase1” i “key1”, generujemy właściwy klucz jak w funkcji send. Deszyfrujemy wiadomość.

Generujemy bazy i przesyłamy je do pierwszego klienta kanałem klasycznym, u nas po prostu zapisując do jego folderu plik “Bbase1”. W ten sposób pierwszy klient ma wszystkie pliki potrzebne do wysłania, albo odebrania następnym razem wiadomości. Natomiast klient 2 nadpisuje stary plik “Bbase1” nowymi bazami, a pliki “Abase1” i “key1” zamienia miejscami z “Abase2” i “key2”. W ten sposób drugi klient również ma wszytkie pliki, niezależnie, czy następnym razem będzie wysyłał czy otrzymywał wiadomość.

Teraz parę uwag do naszego menagera. Załatwienie wszystkiego tylko dwoma funkcjami send i receive było podyktowane architekturą naszej aplikacji, nie musimy tego robić tylko dwoma funkcjami, ale na szczęście się udało. Ilość znaków w kluczu i ilość baz to 32 000, średnio odpada 50% klucza, więc będziemy mieli średnio 16 000 bitów. Można zwiększyć tą ilość, tutaj ryzykujemy brak kodowania ostatnich znaków wiadomości bliskich maksymalnemu limitowi.

Wszystko odbywa się bez ingerencji serwera. Nasz sposób zaimplementowania BB84 nie ma żadnych systemów przeciwko podsłuchowi.

Requirements:

Python 3.7.2