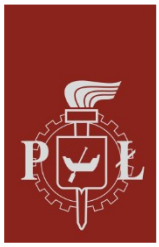


Politechnika Łódzka
Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Jan Wądołowski
230218

PRACA DYPLOMOWA
inżynierska
na kierunku Informatyka Stosowana

Bezpieczny komunikator wykorzystujący metodę szyfrowania SDEx z zastosowaniem funkcji skrótu BLAKE

Instytut Informatyki I72

Promotor: dr inż. Artur Hłobaż

ŁÓDŹ 2024

Spis treści

Rozdział 1

Wstęp

1.1 Streszczenie

Niniejsza praca prezentuje aspekty teoretyczne oraz praktyczne implementacji i wykorzystania algorytmu SDEx w procesie komunikacji między użytkownikami. W celu ilustracji tego zagadnienia została stworzona oraz opisana aplikacja komunikatora mobilnego, w której wiadomości są szyfrowane i deszyfrowane przy użyciu metody SDEx oraz funkcji skrótu BLAKE3.

słowa kluczowe: komunikator, mobilny, szyfrowanie, metoda sdex

1.2 Problematyka i zakres pracy

Niniejsza praca dotyczy zakresu szyfrowania i bezpieczeństwa komunikacji w Internecie.

Głównym przedmiotem pracy jest stworzenie aplikacji mobilnej umożliwiającej wymianę zaszyfrowanych wiadomości tekstowych między jej użytkownikami.

Podjęcie tego tematu jest istotne ze względu na rosnącą liczbę ataków na komunikatory internetowe, mających na celu podszycie się pod osoby lub instytucje, do których adresat ma zaufanie, a w konsekwencji wzmożone zainteresowanie bezpiecznymi środkami komunikacji za pośrednictwem Internetu. Choć istnieją bezpieczne metody szyfrowanej komunikacji zapewniającej uwierzytelnienie zarówno odbiorcy, jak i nadawcy, jednak są one trudne w użyciu dla osób niezaznajomionych z metodami kryptograficznymi (takimi jak kryptografia klucza publicznego) lub nieznających programów umożliwiających stosowanie tych metod. Przez to grupa użytkowników takich narzędzi jest mocno ograniczona.

Istotnym asumptem do wdrożenia aplikacji komunikatora z wykorzystaniem szyfrowania SDEx oraz funkcji skrótu BLAKE w wersji 3 jest także wysoka wydajność szyfrowania przy pomocy tych technologii oraz skalowalność tej wydajności w środowisku przetwarzania współbieżnego, które jest obecnie powszechne w nowoczesnych telefonach komórkowych.

Pewne znaczenie w podejściu do tematu ma ogromna popularność platform społecznościowych umożliwiających publikowanie zdjęć dla wybranych grup użytkowników (na przykład znajomi, znajomi znajomych lub wszyscy zarejestrowani użytkownicy platformy). Wykorzystując tę funkcjonalność użytkownicy stworzonego komunikatora mogą publikować swoje klucze publiczne w postaci wygenerowanego przez aplikację obrazka z kodem QR w galeriach zdjęć na swoich profilach, tym samym udostępniając je swoim znajomym i zachęcając do kontaktu przy pomocy stworzonego komunikatora.

Efektem pracy jest działająca aplikacja mobilna (aplikacja kliencka) i aplikacja serwerowa obsługująca przekazywanie wiadomości między użytkownikami oraz wnioski wyciągnięte z pracy nad wymienionymi produktami.

1.3 Cele pracy

1. Stworzenie działającego komunikatora tekstowego na urządzenia mobilne.
2. Stworzenie oprogramowania serwerowego umożliwiającego uwierzytelnianie użytkowników oraz wymianę wiadomości pomiędzy klientami mobilnymi.
3. Ocena możliwości wdrożenia zaimplementowanego rozwiązania do środowiska produkcyjnego.

1.4 Analiza wymagań funkcjonalnych aplikacji

1.4.1 Dla aplikacji mobilnej

1. Aplikacja obsługiwana przez klienta końcowego (użytkownika aplikacji) winna być w stanie samodzielnie (tzn. bez wymiany informacji z serwerem) obsługiwać proces szyfrowania i deszyfrowania wiadomości przekazywanych do serwera i od niego otrzymywanych.
2. Aplikacja ma odpowiadać za generowanie pary kluczy RSA (prywatnego i publicznego) służących do kryptografii klucza publicznego (kryptografii asymetrycznej).
3. Aplikacja ma być w stanie generować kody QR zawierające klucz publiczny użytkownika oraz eksportować je w formie obrazu rastrowego.
4. Aplikacja musi być wstanie skanować obrazki z takimi kodami QR przy pomocy aparatu, w jaki wyposażony jest telefon komórkowy klienta i wczytywać zakodowany w tym kodzie klucz publiczny RSA innego użytkownika.
5. Aplikacja musi posiadać możliwość rejestracji i logowania klienta, zarówno lokalnie, czyli uwierzytelnienie i autoryzacja dające dostęp do funkcjonalności aplikacji i danych przechowywanych w jej pamięci, jak i zdalnie, czyli uwierzytelnienia użytkownika z danym loginem i kluczem publicznym na serwerze oraz uzyskania na nim uprawnień do niektórych chronionych funkcjonalności, takich jak wymiana wiadomości.
6. Wymagana jest możliwość zmiany informacji użytkownika oraz jego kontaktów - ich nazw oraz kluczy a także możliwość dodawania i usuwania kontaktów.
7. Aplikacja musi być w stanie nawiązywać połączenie internetowe z serwerem i wysyłać oraz odbierać od niego wiadomości w sposób bezpieczny (tzn. szyfrując wrażliwe dane).

1.4.2 Dla aplikacji serwerowej

1. Aplikacja serwerowa winna działać w sposób bezobsługowy, to jest po jej wdrożeniu na serwerze i uruchomieniu powinna bez konieczności czynnej obsługi administratora obsługiwać komunikację z klientami mobilnymi zgodnie z przewidzianymi scenariuszami działania.

2. Aplikacja musi działać w sposób scentralizowany tzn. jedna instancja aplikacji musi być w stanie obsługiwać równocześnie, w granicach przewidzianego obciążenia, wielu klientów mobilnych.

3. Aplikacja ma komunikować się z klientami zewnętrznymi (tzn. oprogramowaniem operującym na innych urządzeniach) w sposób bezpieczny, czyli przesyłając wrażliwe dane w formie zaszyfrowanej.

4. Aplikacja powinna być możliwie bezstanowa, tzn. przechowywać minimalną ilość informacji o użytkownikach i dane jedynie aktywnych sesji. Dane o zakończonych sesjach oraz historia wymienianych wiadomości ma nie być zapamiętywana.

5. Serwer powinien być w stanie uwierzytelniać użytkownika na podstawie jego loginu weryfikując przy tym, że użytkownik z zarejestrowanym loginem i kluczem publicznym posiada pasujący do niego klucz prywatny. Klucz prywatny użytkownika nie może jednak być przechowywany (ani nawet wysyłany) do serwera.

6. Autoryzacja ma być możliwa zarówno w procesie początkowego rejestrowania użytkownika na serwerze, jak i przy kolejnych żądaniach logowania na serwerze.

7. Serwer musi autoryzować użytkownika wykonującego niektóre zastrzeżone operacje (takie jak wysyłanie wiadomości), tzn. sprawdzać czy połączony użytkownik wykonujący zapytanie jest zarejestrowany na serwerze.

8. Serwer ma umożliwiać przesyłanie wiadomości między klientami, tzn. przekazywać otrzymaną od jednego z klientów wiadomość do klienta wskazanego jako odbiorca wiadomości (i tylko do niego).

9. Serwer ma nie ingerować w żaden sposób w proces deszyfrowania wiadomości przez klienta-adresata wiadomości.

10. Serwer ma umożliwiać sprawdzanie stanu połączenia (połączony / nie połączony) wybranego klienta na żądanie innego klienta.

11. Serwer ma umożliwiać zmianę klucza publicznego danego klienta na żądanie tego klienta, jednak w taki sposób, aby żaden klient nie mógł wykonać tej operacji dla osoby trzeciej.

12. Serwer powinien realizować na żądanie użytkownika weryfikacji, czy dany klucz publiczny jest zarejestrowany na serwerze.

Rozdział 2

Szyfrowanie treści metodą SDEx (część teoretyczna)

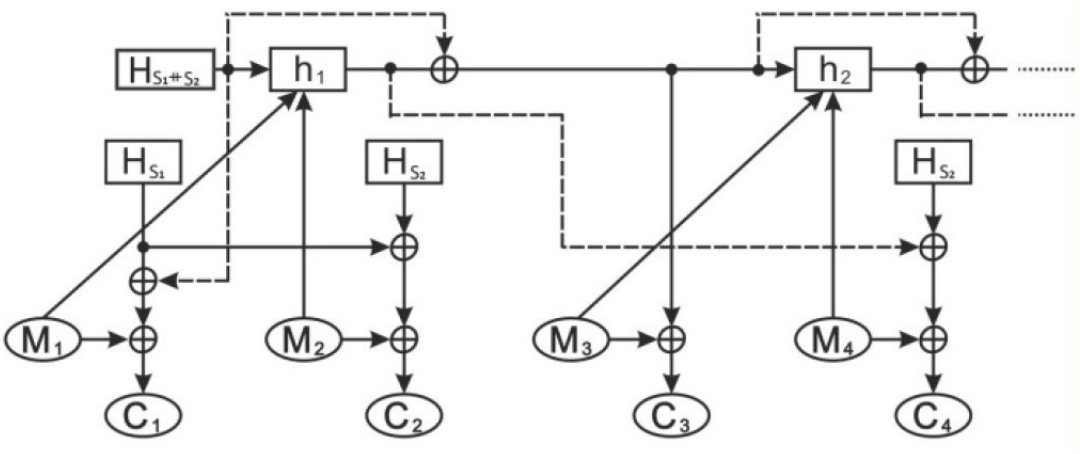
2.1 Opis metody szyfrowania i deszyfrowania SDEx

Na rysunkach 2.1 i 2.2 oraz w równaniach 2.1 - 2.11 użyto oznaczeń:

- $\{M_1, M_2, \ldots, M_i\}$ - bloki tekstu jawnego,
- $\{C_1, C_2, \ldots, C_i\}$ - bloki tekstu zaszyfrowanego,
- $\{h_1, h_2, \ldots, h_k\}$ - dane iteracje hashu,
- H_{S1} - hash z pierwszego klucza sesji,
- H_{S2} - hash z drugiego klucza sesji,
- $H_{S1 \# S2}$ - hash ze złączonych pierwszego i drugiego klucza sesji,
- $\oplus$ - operacja XOR,
- $\#$ - złączenie (konkatenacja) dwóch łańcuchów tekstowych,

Szyfrowanie metodą SDEx odbywa się zgodnie z równaniami 2.1 - 2.7 i ma formę przedstawioną na Rysunku 2.1:

$$C_1 = M_1 \oplus H_{S1} \oplus H_{S1 \# S2} \tag{2.1}$$
$$C_2 = M_2 \oplus H_{S1} \oplus H_{S2} \tag{2.2}$$
$$C_{2k+1} = M_{2k+1} \oplus h_k \oplus h_{k-1} \qquad k \geq 1 \tag{2.3}$$
$$C_{2k+2} = M_{2k} \oplus H_{S2} \oplus h_k \qquad k \geq 1 \tag{2.4}$$
$$h_1 = hash(H_{S1 \# S2}; M_1 \# M_2) \tag{2.5}$$
$$h_2 = hash((h_1 \oplus H_{S1 \# S2}); M_3 \# M_4) \tag{2.6}$$
$$h_k = hash((h_{k-1} \oplus h_{k-2}); M_{2k-1} \# M_{2k}) \qquad k \geq 3 \tag{2.7}$$



Rysunek 2.1: Diagram algorytmu szyfrowania SDEx

Źródło: dr inż. Artur Hłobaż

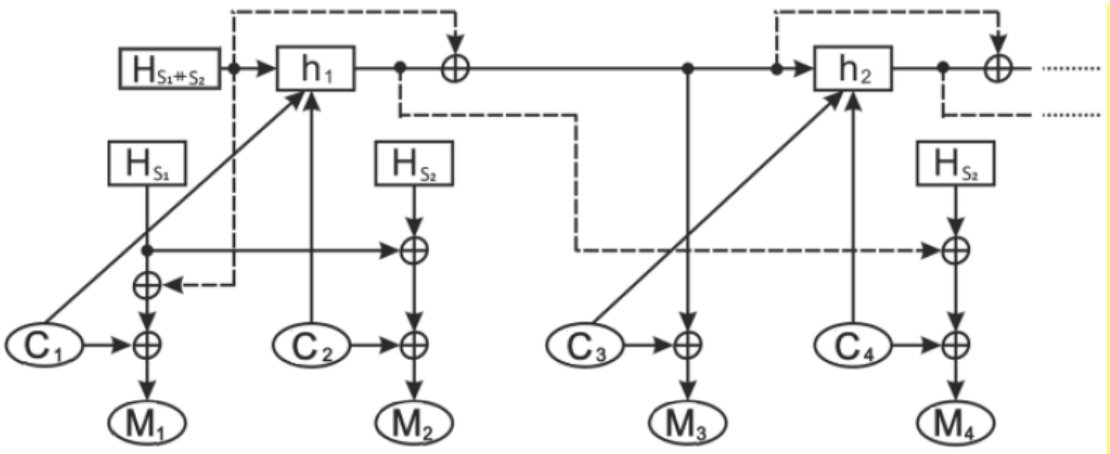
Deszyfrowanie metodą SDEx odbywa się zgodnie z równaniami 2.8 - 2.11 i ma formę przedstawioną na Rysunku 2.2:

$$M_1 = C_1 \oplus H_{S1} \oplus H_{S1\#S2} \tag{2.8}$$

$$M_2 = C_2 \oplus (H_{S1} \oplus H_{S2}) \tag{2.9}$$

$$M_{2k+1} = C_{2k+1} \oplus h_k \oplus h_{k-1} \qquad k \geq 1 \tag{2.10}$$

$$M_{2k+2} = C_{2k} \oplus H_{S2} \oplus h_k \qquad k \geq 1 \tag{2.11}$$



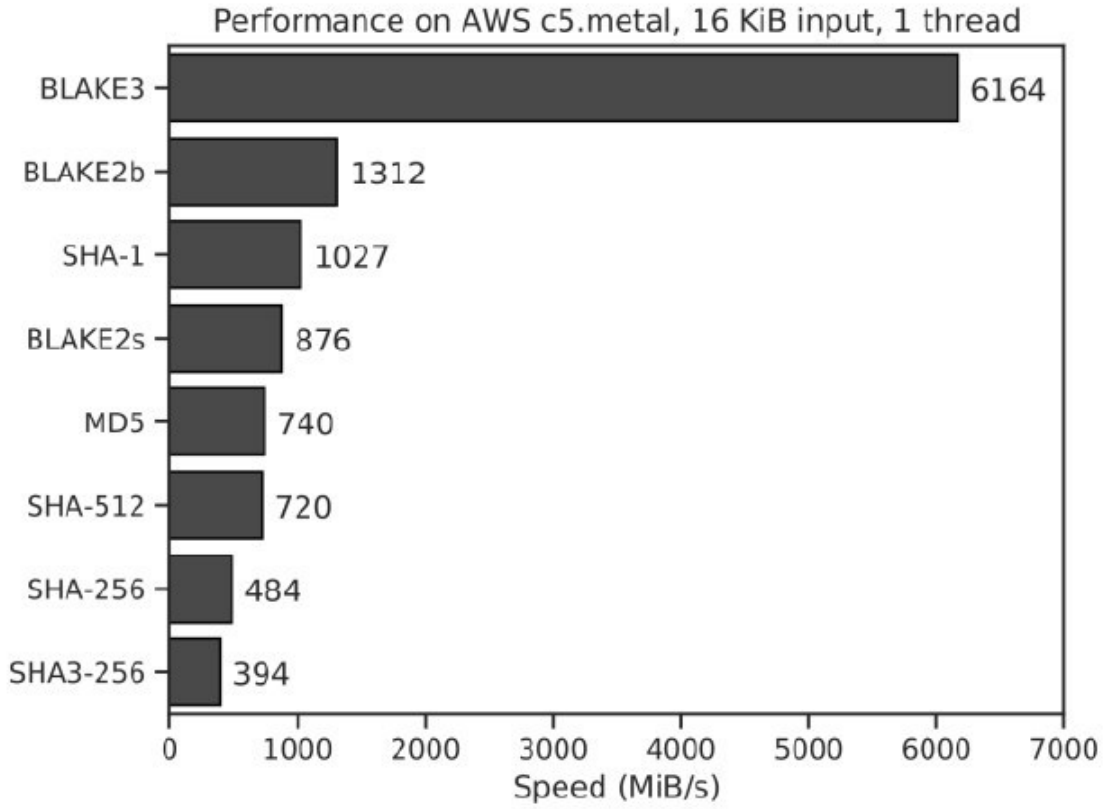
Rysunek 2.2: Diagram algorytmu deszyfrowania SDEx

Źródło: dr inż. Artur Hłobaż

Zastosowana została uproszczona metoda SDEx bazująca na publikacji *Analysis of enhanced SDEx method*[14] po uzgodnieniu z promotorem.

2.2 Wybór funkcji skrótu BLAKE 3 do użycia z algorytmem szyfrującym SDEx

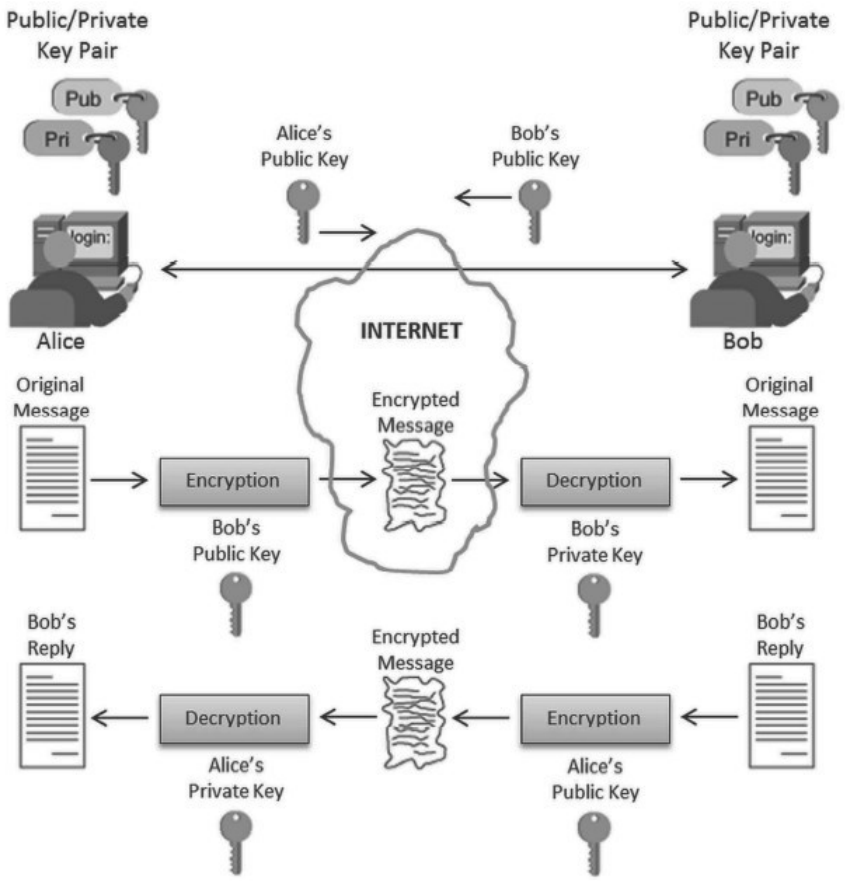
Ze wzorów opisujących proces szyfrowania i deszyfrowania metodą SDEx widać, że głównym czynnikiem decydującym o wydajności algorytmu jest wydajność wykorzystanej funkcji hashującej. Publikacja *Analysis of the Possibility of Using Selected Hash Functions Submitted for the SHA-3 Competition in the SDEx Encryption Method*[7] przedstawia porównanie efektywności obliczeniowej wybranych popularnych algorytmów hashujących. Wynika z niego, że funkcja skrótu BLAKE 3 oferuje najwyższą wydajność w przeliczeniu na jeden wątek procesora, a także lepszy margines bezpieczeństwa od funkcji hashujących z grupy Grøstl, JH czy Keccak.



Rysunek 2.3: Wydajności obliczeniowe wybranych funkcji skrótu[7]

2.3 Szyfrowanie i podpisywanie metodą klucza publicznego

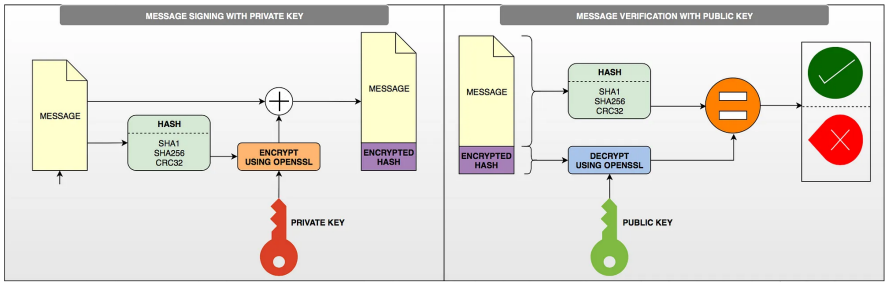
Do nawiązania komunikacji z serwerem, a w dalszej kolejności także wymiany informacji potrzebnych do realizacji szyfrowania i deszyfrowania metodą SDEx potrzebna jest bezpieczna wymiana pewnych informacji między klientem a serwerem oraz między klientami komunikującymi się ze sobą. Do tego celu wykorzystano metodę kryptografii asymetrycznej w oparciu o klucze RSA. Proces komunikacji z użyciem tej techniki kryptograficznej prezentuje Rysunek 2.4.



Rysunek 2.4: Proces wymiany wiadomości z użyciem kryptografii asymetrycznej[10]

Z użyciem tej metody wymieniane są między klientami wygenerowane przez nich części klucza sesji $S1$ i $S2$.

Kryptografia asymetryczna umożliwia także podpisywanie wiadomości kluczem prywatnym nadawcy, aby odbiorca mógł zweryfikować autentyczność i niezmienność przyjętej wiadomości. Rysunek 2.5 przedstawia proces podpisywania wiadomości i jej weryfikacji. W kontekście niniejszej publikacji jest to wykorzystywane do uwierzytelnienia klienta na serwerze.



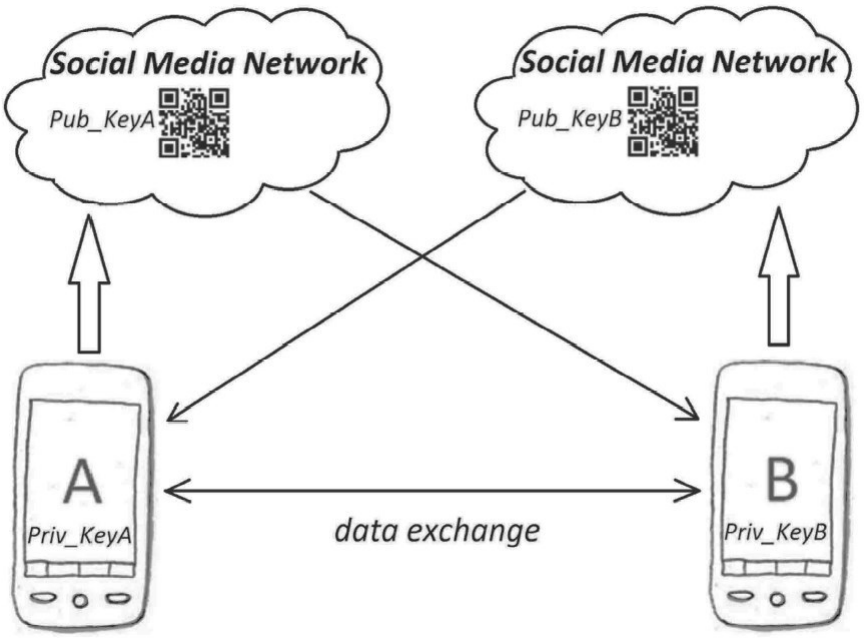
Rysunek 2.5: Podpisywanie wiadomości i jej weryfikacja z użyciem kryptografii asymetrycznej[2]

2.4 Rozpowszechnianie kluczy publicznych użytkowników poprzez kody QR

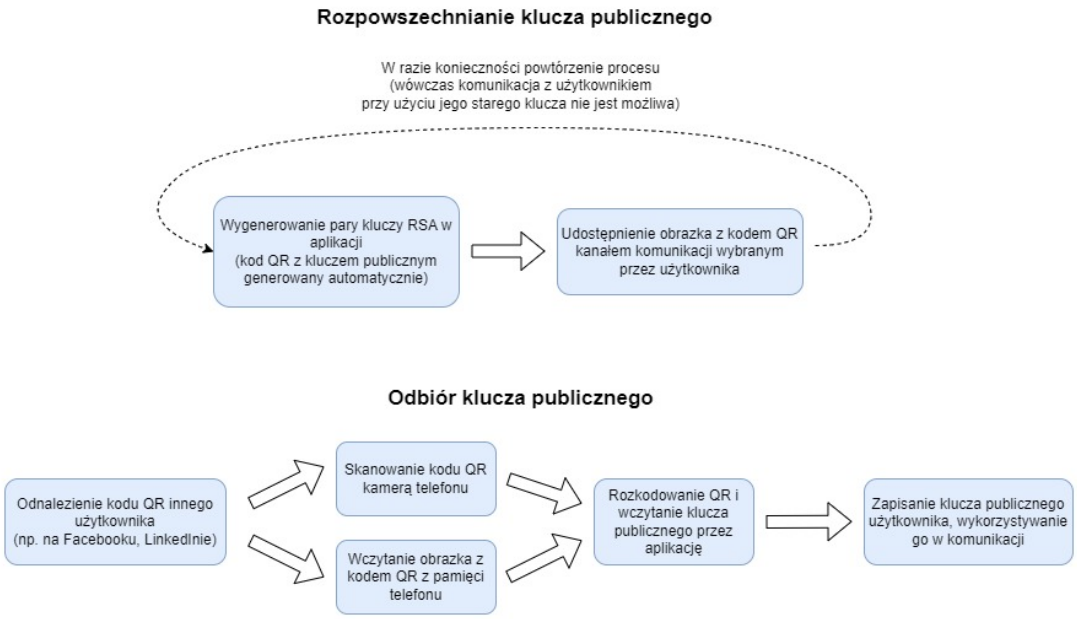
Klucze publiczne wygenerowane lub wczytywane przez klientów aplikacji, mają wyjściowo formę tekstową, trudną do zapamiętania, a ich rozpowszechnianie metodą kopiowania i wklejania jest mało atrakcyjne dla użytkowników (w porównaniu przykładowo do wymiany numerów telefonów, które są krótsze i łatwiejsze do krótkotrwałego zapamiętania dla człowieka) oraz podatne na błędy, gdyż zawierają one znaki nowej linii, w tym znak nowej linii na końcu tekstu klucza, który łatwo przeoczyć przy kopiowaniu, co może powodować błąd przy próbie zaszyfrowania wiadomości takim niepełnym kluczem.

Wybraną metodą rozpowszechniania kluczy publicznych jest zakodowanie ich do postaci kodu QR. Posiadają one bardziej atrakcyjną wizualnie formę obrazka, który łatwo można udostępnić na przykład na platformie społecznościowej, a także umożliwiają klientowi w prosty sposób wykorzystanie go, aby dodać klucz publiczny innego użytkownika poprzez zeskanowanie obrazka aparatem w telefonie lub wczytanie go z pamięci telefonu. Ponadto standard kodu QR zapewnia korekcję błędów do nawet 30% zakodowanej zawartości[3].

Wymianę kluczy publicznych tą metodą w oparciu o platformę społecznościową przedstawiają rysunki 2.6 oraz 2.7.



Rysunek 2.6: Wymiana kodów QR z kluczami publicznymi w oparciu o platformę społecznościową[10]



Rysunek 2.7: Wymiana kodów QR z kluczami publicznymi

Źródło: opracowanie własne

2.5 Wybór technologii do realizacji części praktycznej pracy

2.5.1 Technologie aplikacji mobilnej

W celu realizacji niniejszej pracy potrzeba było stworzyć aplikację mobilną uruchamianą na telefonie komórkowym (smartfonie). Istnieje wiele popularnych technologii umożliwiających stworzenie takiego oprogramowania, które dawałyby możliwość implementacji założeń opisanych w sekcji 1.4.1. Technologie te można podzielić na natywne (*native apps*), natywne między platformowe (*native cross-platform*), sieciowe (*web apps*), progresywne aplikacje sieciowe (*progressive web applications*, PWA) oraz łączące cechy aplikacji natywnych i sieciowych - czyli hybrydowe[9][17].

Aplikacje natywne z reguły są wydajniejsze i bardziej energooszczędne od ich webowych odpowiedników[8], a do tego oferują pełniejszy dostęp do funkcjonalności urządzenia, na którym pracują (na przykład możliwość korzystania z kamery urządzenia, jego pamięci masowej czy wbudowanej szyfrowanej bazy danych do przechowywania certyfikatów i haseł).

Istotną kwestią przy wyborze technologii jest także dojrzałość danego "ekosystemu", czyli dostępność bibliotek ułatwiających tworzenie oprogramowania, aktualność i zgodność tych bibliotek ze współczesnymi standardami bezpieczeństwa oraz aktualnymi wersjami interfejsów programowania aplikacji (ang. *application programming interface*, API) zgodnymi z nowszymi wersjami systemów operacyjnych, na których działają współczesne telefony, a także dostępność narzędzi wspomagających procesy testowania, debugowania i budowania aplikacji.

Do realizacji części praktycznej niniejszej pracy wykorzystany został framework React Native umożliwiający tworzenie natywnych aplikacji między platformowych. Z użyciem tego frameworka kod napisany w języku JavaScript jest kompilowany do kodu natywnego dla platformy docelowej[12], co przy wzięciu pod uwagę pewnych różnic wynikających z odmienności platform mobilnych (różne funkcjonalności oraz niejednolite API tych platform) pozwala stworzyć kod działający na urządzeniach różnych producentów w sposób jednakowy lub zbliżony.

Do wyboru tej technologii przyczyniły się:

- szeroka dostępność bibliotek,
- wsparcie i rozwój tej technologii przez rozpoznawalną i doświadczoną w branży IT firmę (Meta),
- możliwość tworzenia natywnych modułów pisanych w językach właściwych dla danej platformy (dla Androida jest to Java i Kotlin, dla iOS - Swift oraz ObjectiveC), co w przypadku braku implementacji pewnych funkcji tych platform w ramach frameworka i bibliotek umożliwiałoby ich własnoręczne zaimplementowanie),
- wydajność aplikacji tworzonych z użyciem tego frameworka jest zbliżona do aplikacji tworzonych w językach natywnych dla wybranej platformy (np. w Javie)[13],
- rozpowszechniająca się adopcja React Native w produktach komercyjnych[4]

2.5.2 Wybór technologii backendowej do implementacji aplikacji serwerowej

Do realizacji aplikacji serwerowej należało wybrać technologie umożliwiające zarówno komunikację webową (za pośrednictwem websocketów) z klientami mobilnymi, jak również mogącą komunikować się z bazą danych, co wynika z założeń funkcjonalnych opisanych w sekcji 1.4.2.

Istotną kwestią przy wyborze tych technologii była obsługa asynchronicznej komunikacji z pojedynczym oraz z wieloma klientami. Również dla aplikacji serwerowej istotnym czynnikiem była dostępność bibliotek, które implementują wymagane funkcjonalności oraz aktywny rozwój przez ich autorów. Ze względu na te wymagania, a także wcześniejszą dobrą znajomość tych technologii przez autora niniejszego opracowania wybrany został język Python oraz framework FastAPI działający na serwerze Uvicorn. Na ten wybór wpłynęła przede wszystkim leżąca u podstaw frameworka FastAPI i prosta w programowaniu asynchroniczność obsługi żądań (requestów), rozwiązania bazujące na aktualnych standardach języka Python oraz dość wysoka wydajność tego frameworka i Uvicorna[16]. Korzyścią wynikającą z wyboru języka Python jest fakt, że posiada on wbudowane w bibliotekę standardową wsparcie dla bazy danych SQLite, która została wykorzystana do przechowywania danych o klientach.

2.5.3 Wybór protokołu łączności klientów mobilnych z serwerem

Przy wyborze protokołu łączności między urządzeniami klientów a serwerem brano pod uwagę protokół HTTP z architekturą REST (ang. *representational state transfer*) i protokół WebSocket. Po przeanalizowaniu wymagań funkcjonalnych, jakie musi spełniać aplikacja oraz różnic w zastosowaniu tych rozwiązań[15][1] jednoznacznie wybrana została

komunikacja w oparciu o protokół WebSocket. Przede wszystkim WebSocket lepiej nadaje się do obsługi komunikacji wywołanej zdarzeniami (*event-based*), która jest typowa dla aplikacji typu czat. Dzięki temu można w prosty sposób zaimplementować przekazywanie wiadomości w obie strony (full-duplex), to jest od klienta do serwera i odwrotnie, a także trójstronną komunikację (klient 1 - serwer - klient 2), a przy tym z minimalnymi opóźnieniami pomiędzy wysłaniem i doręczeniem wiadomości. Chcąc odwzorować ten mechanizm w oparciu o protokół HTTP trzeba byłoby wykonywać cykliczne zapytania do serwera w krótkich odstępach czasu, aby uzyskać informacje zwrotne z przebiegu komunikacji.

Rozdział 3

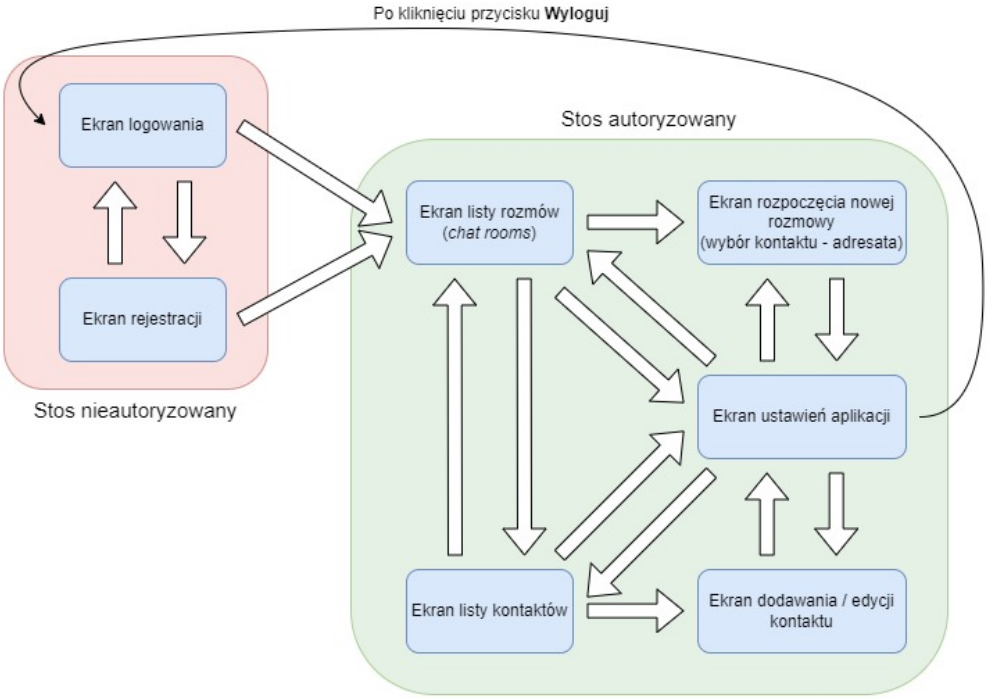
Implementacja komunikatora szyfrującego wiadomości metodą SDEx

3.1 Architektura rozwiązania

Komunikator powstał w oparciu o model klient - serwer, gdzie serwer odpowiedzialny jest za wymianę wiadomości między użytkownikami, uwierzytelnianie klientów przy nawiązywaniu połączenia oraz weryfikację uprawnień przy wykonywaniu zapytań do niego. Natomiast aplikacja kliencka odpowiada za wczytywanie klucza publicznego z kodu QR, generowanie kodu QR z kluczem publicznym klienta, szyfrowanie nadawanych wiadomości oraz deszyfrowanie odbieranych wiadomości, a także przechowywanie historii czatu w pamięci lokalnej. Klient mobilny jest też odpowiedzialny za autoryzację użytkownika w ramach aplikacji (możliwy jest dostęp do aplikacji, historii rozmów czy ustawień bez autoryzacji ze strony serwera).

3.2 Mapa oraz wygląd widoków aplikacji mobilnej

Rysunek 3.1 przedstawia mapę widoków (ekranów) oraz możliwych przejść między nimi. Implementacja aplikacji dzieli ekrany na 2 stosy - stos autoryzowany i nieautoryzowany. Dzięki temu podziałowi uwierzytelniony użytkownik nie ma możliwości przypadkowego powrotu do ekranu logowania np. po naciśnięciu przycisku lub wykonaniu gestu "cofnij"na swoim urządzeniu, z kolei użytkownik niezalogowany nie będzie miał możliwości pominięcia logowania lub rejestracji i przedostania się do ekranów (a co za tym idzie funkcjonalności) dostępnych jedynie po zalogowaniu.



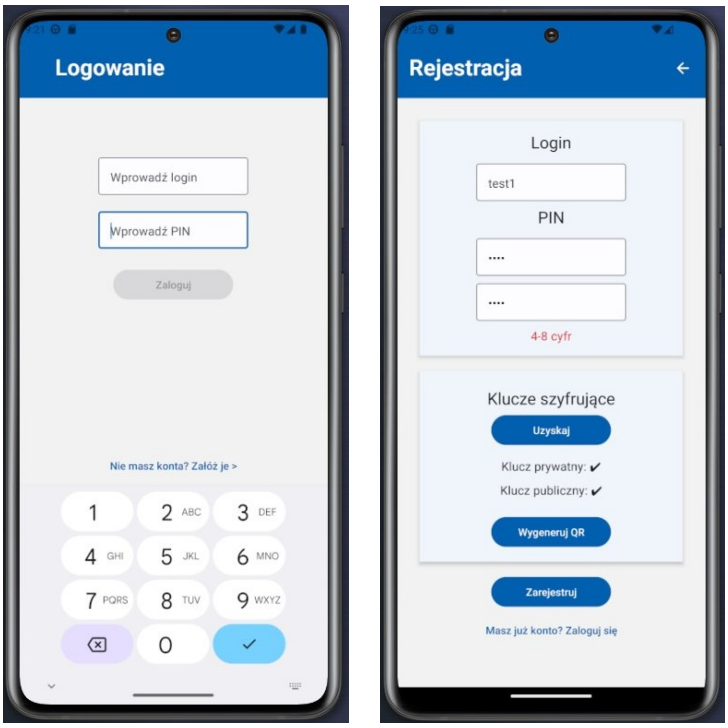
Rysunek 3.1: Mapa widoków i możliwych przejść między nimi

Źródło: opracowanie własne

Wygląd poszczególnych widoków aplikacji przedstawiają zrzuty ekranów na Rysunkach 3.2-3.6

Rysunek 3.2: Ekran czatu

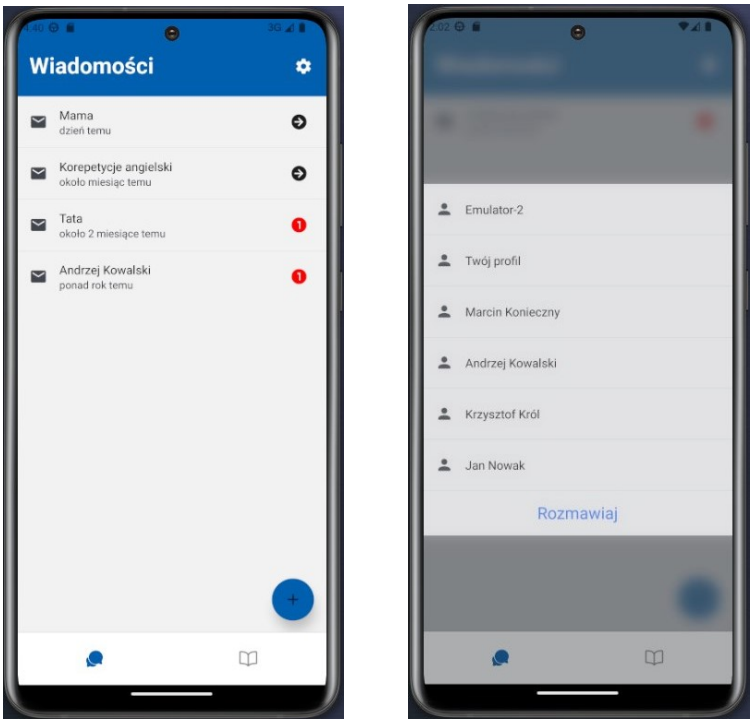
Źródło: opracowanie własne



(a) Ekran logowania (b) Ekran rejestracji

Rysunek 3.3: Zrzuty ekranów aplikacji mobilnej

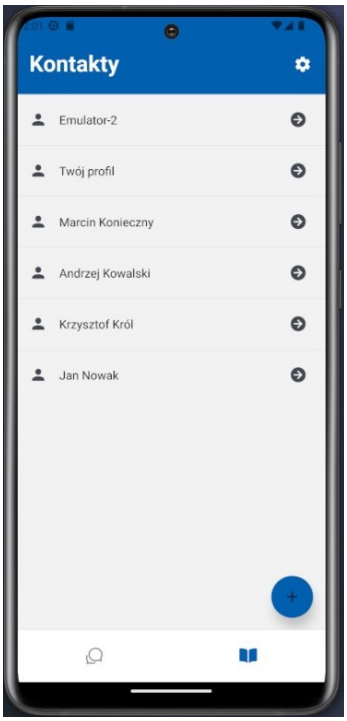
Źródło: opracowanie własne



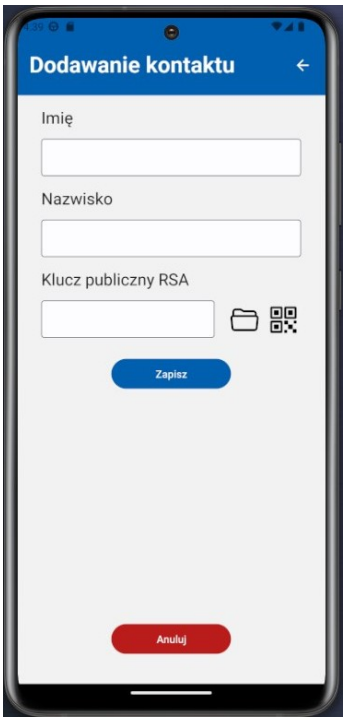
(a) Ekran listy rozmów (b) Ekran tworzenia rozmowy

Rysunek 3.4: Zrzuty ekranów aplikacji mobilnej

Źródło: opracowanie własne



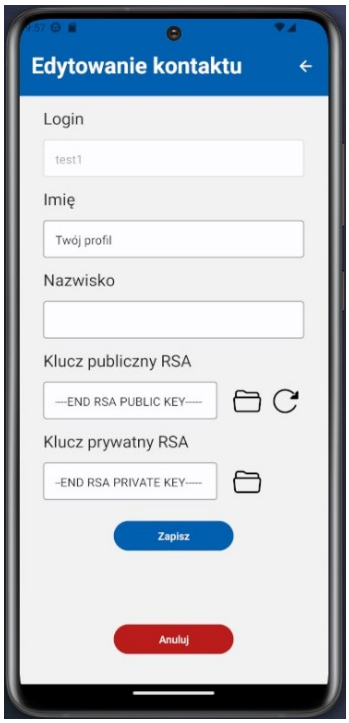
(a) Ekran listy kontaktów



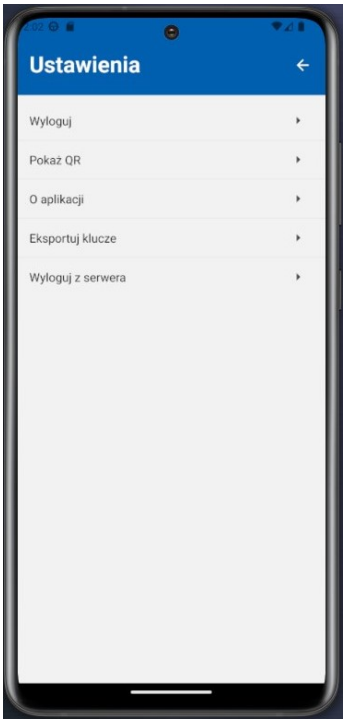
(b) Ekran tworzenia kontaktu

Rysunek 3.5: Zrzuty ekranów aplikacji mobilnej

Źródło: opracowanie własne



(a) Ekran edytowania kontaktu



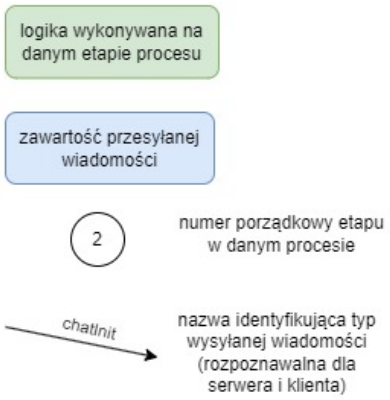
(b) Ekran ustawień aplikacji

Rysunek 3.6: Zrzuty ekranów aplikacji mobilnej

Źródło: opracowanie własne

3.3 Realizacja funkcjonalności aplikacji

Implementacja głównych funkcjonalności aplikacji wiąże się z szeregiem działań jakie musi wykonać klient mobilny oraz serwer. Poniżej przedstawione i opisane zostały główne funkcje aplikacji w formie diagramów procesów.

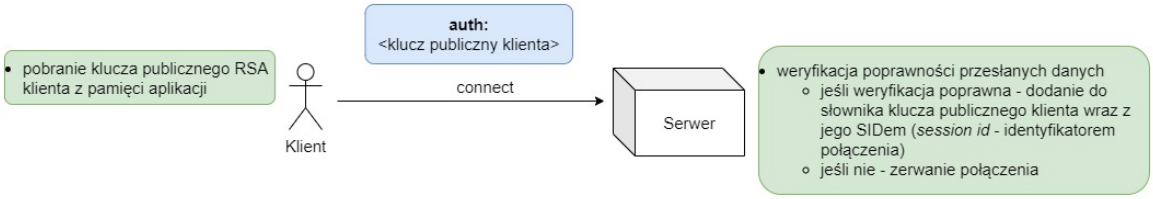


Rysunek 3.7: Oznaczenia symboli na diagramach procesów

Źródło: opracowanie własne

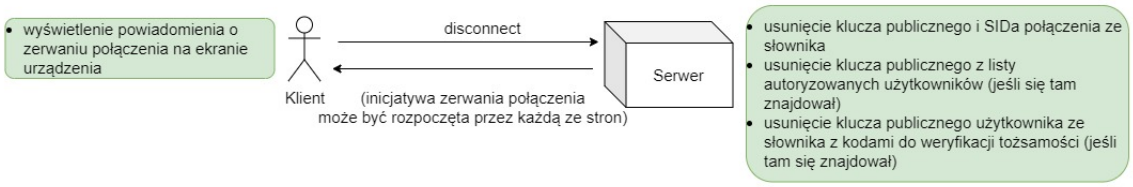
3.3.1 Nawiązywanie i zrywanie połączenia z serwerem

Przy nawiązywaniu połączenia między aplikacją mobilną a serwerem generowany jest unikatowy identyfikator połączenia sesji (ang. *session ID*, SID). Zadaniem serwera jest zapamiętanie powiązania klucz klienta - SID, jest to potrzebne przy realizacji innych procesów. Serwer aktualizuje dane w odpowiedzi na zmianę sytuacji (np. zerwanie połączenia i ponowne jego nawiązanie skutkuje uaktualnieniem powiązania klucz klienta - SID, a definitywne zerwanie połączenia - usunięciem tego wpisu z pamięci serwera).



Rysunek 3.8: Proces nawiązywania połączenia klienta z serwerem

Źródło: opracowanie własne



Rysunek 3.9: Proces zrywania połączenia klienta z serwerem

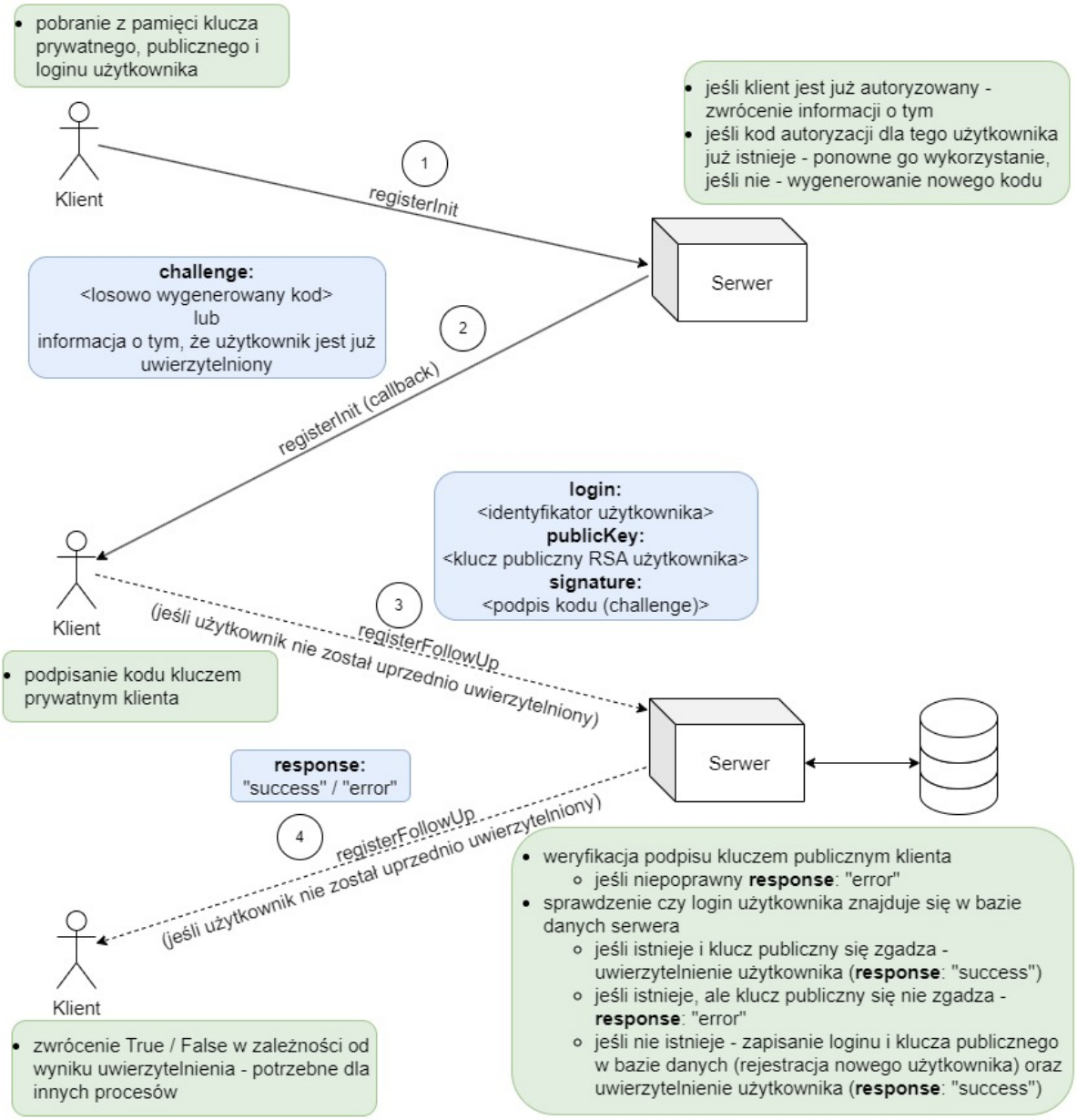
Źródło: opracowanie własne

3.3.2 Uwierzytelnianie i rejestracja użytkownika

W procesie uwierzytelniania wykorzystywane jest podpisywanie i weryfikowanie podpisu wiadomości kluczami RSA. Serwer generuje losowy ciąg znaków i przekazuje go klientowi, który podpisuje tę wiadomość przy pomocy swojego klucza prywatnego oraz odsyła ją do serwera. Następnie serwer przy pomocy klucza publicznego klienta weryfikuje poprawność wiadomości. Gwarantuje to, że użytkownik posiadający dany klucz publiczny posiada też odpowiadający mu klucz prywatny.

Ponadto serwer pobiera z bazy danych login i klucz publiczny, którymi identyfikuje się klient. W przypadku gdy w bazie danych nie ma tych wpisów, ale weryfikacja podpisu przebiegła pomyślnie następuje rejestracja użytkownika w bazie danych, po tym każda ponowna próba rejestracji na serwerze z tym loginem, ale innym kluczem publicznym lub z tym samym kluczem, ale innym loginem będzie odrzucana.

Użytkownik ma możliwość późniejszej zmiany klucza publicznego, ale jedynie w sytuacji gdy uzyskał autoryzację na serwerze w ramach danego połączenia.

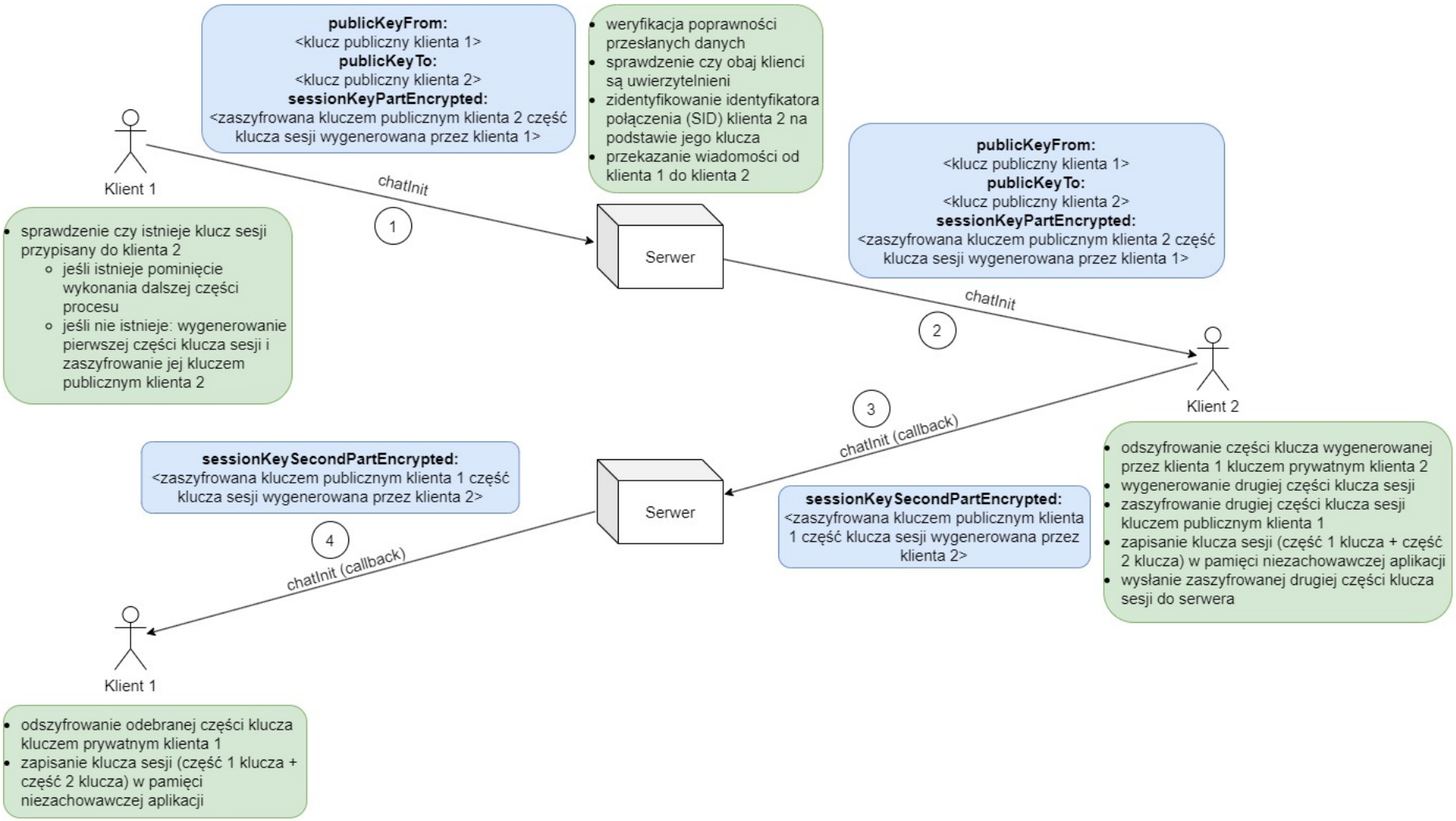


Rysunek 3.10: Proces uwierzytelniania klienta na serwerze

Źródło: opracowanie własne

3.3.3 Nawiązywanie komunikacji między użytkownikami

Wykorzystanie algorytmu SDEx do szyfrowania wiadomości między użytkownikami wymaga wymiany między nimi części klucza sesji $S1$ i $S2$. serwer pełni tu jedynie rolę pośrednika w wymianie wiadomości, ale sam nie ma wglądu w części klucza (są one przekazywane w postaci zaszyfrowanej przy pomocy kluczy RSA użytkowników) ani nie uczestniczy w szyfrowaniu i deszyfrowaniu wiadomości między użytkownikami, jest to wykonywane przez ich aplikacje klienckie.



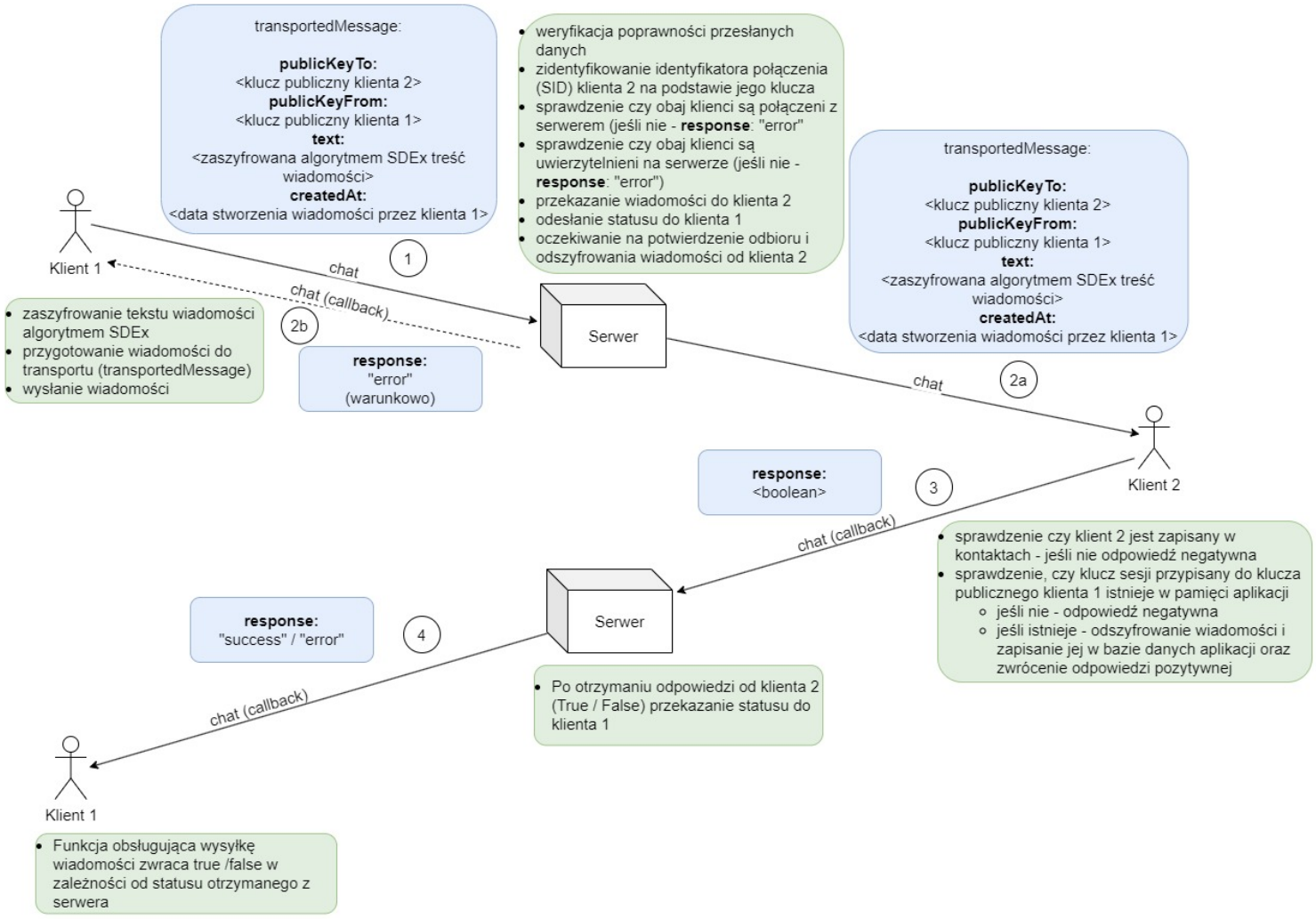
Rysunek 3.11: Proces nawiązywania komunikacji między użytkownikami

Źródło: opracowanie własne

3.3.4 Przekazanie wiadomości między użytkownikami

Po uprzedniej inicjalizacji połączenia między klientami (Rozdział 3.3.3) użytkownicy posiadają uwspólniony klucz sesji, dzięki któremu są w stanie szyfrować i deszyfrować algorytmem SDEx wysyłane między sobą wiadomości.

Proces szyfrowania / deszyfrowania oraz przygotowania wiadomości do wysyłki (która oprócz treści zawiera metadane takiej jak klucz publiczny odbiorcy i adresata oraz datę utworzenia) przebiega w aplikacji mobilnej, natomiast serwer pośredniczy w przekazaniu tak przygotowanej wiadomości między użytkownikami (gdyż nie mają oni bezpośredniego połączenia między sobą). Serwer odpowiada także za zapewnienie autoryzacji obu klientów do wysyłki i odbioru wiadomości, sprawdza ich status dostępności, a na końcu zwraca potwierdzenie doręczenia i poprawnego odszyfrowania wiadomości przez odbiorcę do jej nadawcy. Na etapach oznaczonych na Rysunku 3.12 numerami 2a i 3 jeśli nastąpi zerwanie połączenie Klienta 2 (odbiorcy wiadomości) z serwerem wiadomość zostaje uznana za niedostarczoną i serwer informuje o tym Klienta 1 (nadawcę) odpowiednio w kroku 2b lub 4. Komunikacja serwera z klientami toleruje opóźnienia w uzyskaniu odpowiedzi od odbiorców (tzw. *timeout*) do 60 sekund. Po tym czasie serwer przestaje nasłuchiwać odpowiedzi i zwraca błąd doręczenia do nadawcy. Nie następuje ponowna próba doręczenia wiadomości przy wystąpieniu timeoutu lub utraty połączenia przez odbiorcę.



Rysunek 3.12: Proces przekazania wiadomości między dwoma użytkownikami

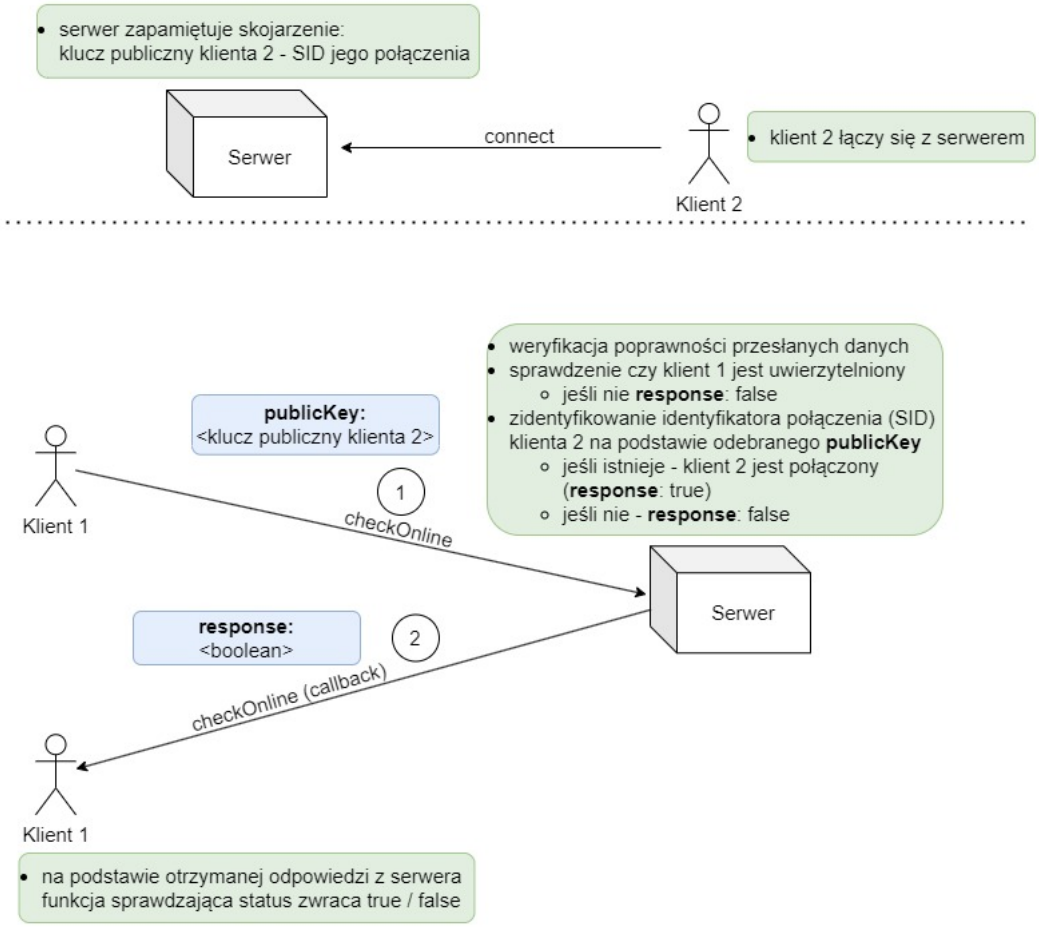
Źródło: opracowanie własne

3.3.5 Sprawdzanie statusu połączenia innego klienta z serwerem oraz poprawności jego klucza publicznego

Te dwie funkcjonalności mają charakter wspomagający dla procesów opisanych w rozdziałach 3.3.3 i 3.3.4. Dzięki ich zastosowaniu aplikacja może informować użytkownika próbującego przekazać wiadomość do jednego ze swoich kontaktów jeśli taka wymiana się nie powiedzie ze względu na to, że odbiorca nie jest połączony z serwerem lub że jego klucz publiczny nie został na nim zarejestrowany. Dzieje się to poprzez wyświetlenie baneru z odpowiednią wiadomością na ekranie czatu w aplikacji mobilnej (Rysunek 3.13).

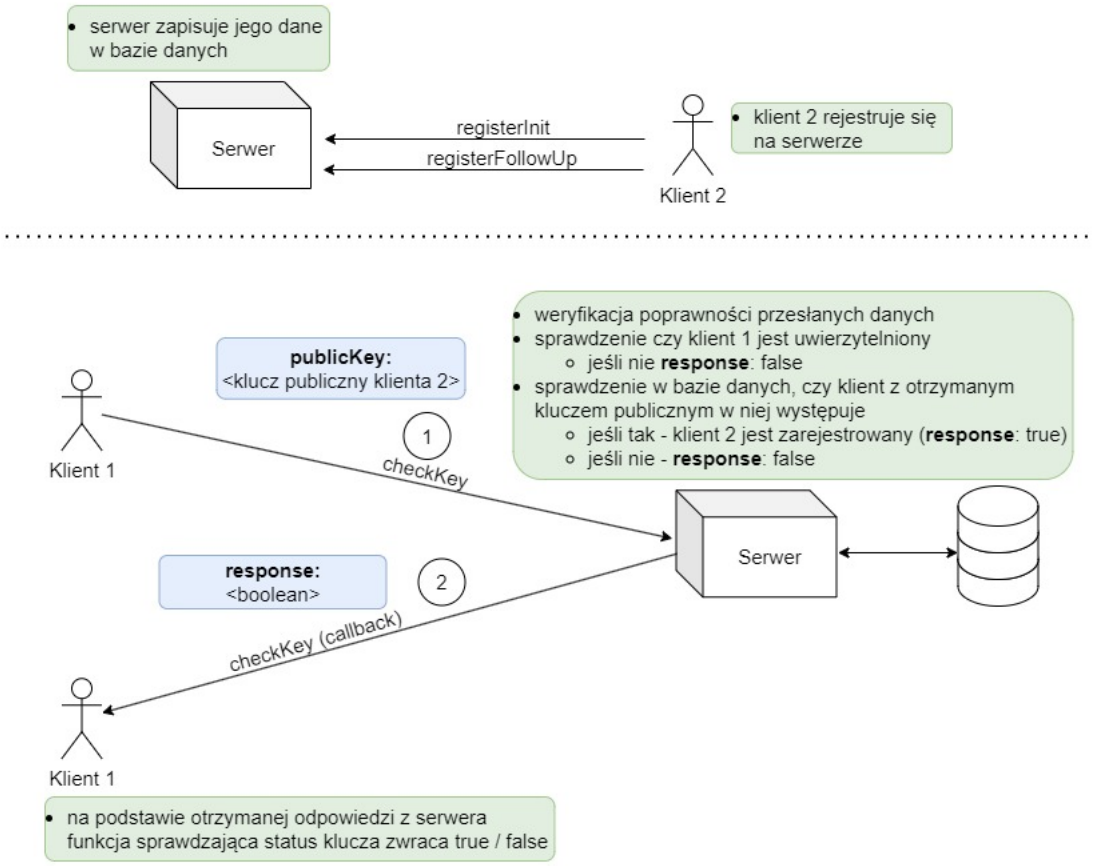
Rysunek 3.13: Baner informujący o niepoprawnym kluczu publicznym kontaktu

Źródło: opracowanie własne



Rysunek 3.14: Proces sprawdzania dostępności klienta na serwerze

Źródło: opracowanie własne



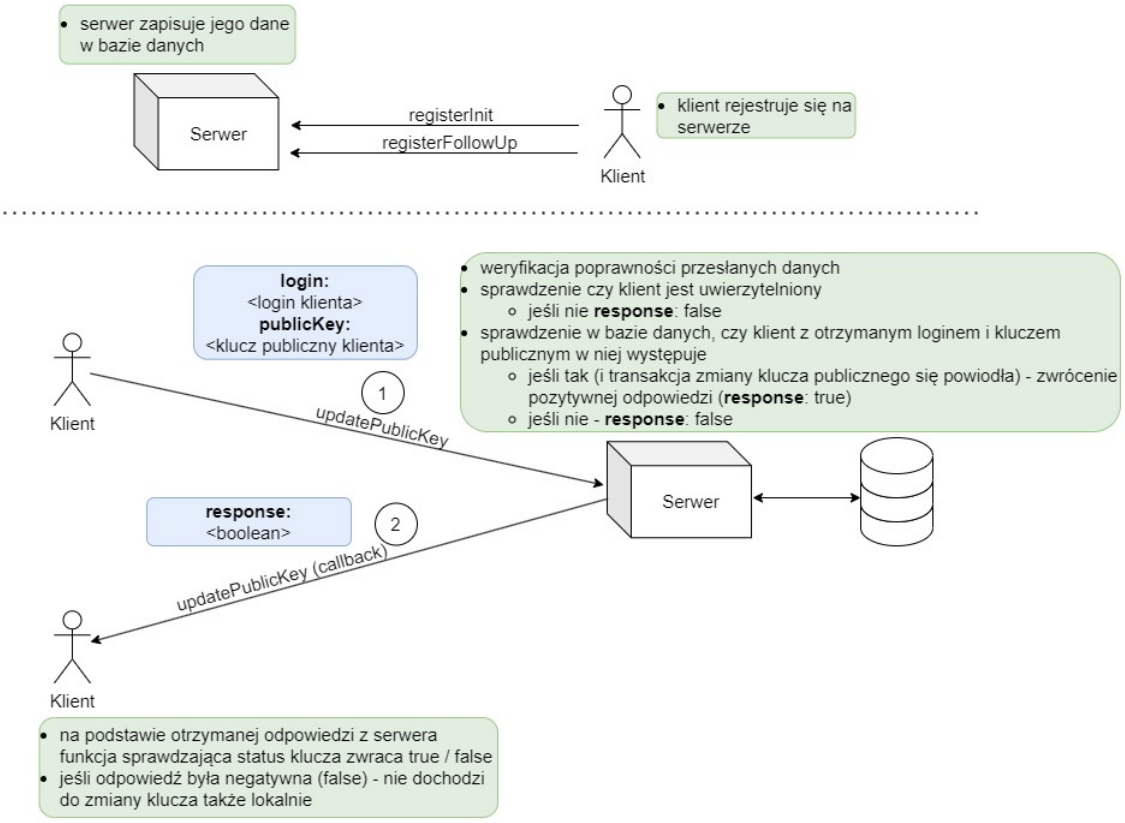
Rysunek 3.15: Proces sprawdzania poprawności klucza publicznego klienta na serwerze

Źródło: opracowanie własne

3.3.6 Zmiana klucza publicznego klienta na serwerze

Zarejestrowany na serwerze użytkownik ma możliwość zmiany swojej pary kluczy RSA. Wykonywane jest to w aplikacji za pomocą formularza przedstawionego na Rysunku 3.6 (a).

W wyniku tego konieczna jest zmiana klucza publicznego także na serwerze. Proces ten przedstawia Rysunek 3.16. Co istotne, w przypadku niepowodzenia zmiany klucza publicznego na serwerze również w aplikacji mobilnej zmiana ta nie zostaje zapisana w pamięci zachowawczej aplikacji, w innym wypadku użytkownik trwale utraciłby możliwość uwierzytelnienia się na serwerze.



Rysunek 3.16: Proces sprawdzania poprawności klucza publicznego klienta na serwerze

Źródło: opracowanie własne

3.4 Wybrane fragmenty kodu aplikacji wraz z omówieniem

Poniżej przedstawione są listingi kodu aplikacji, które wraz z komentarzem mają obrazować sposób implementacji rozwiązania. Są to fragmenty typowe (takie, które w podobnej formie pojawiają się w wielu miejscach w aplikacji) lub kluczowe z perspektywy problematyki podjętego opracowania - szyfrowania metodą SDEx. Z przytaczanych przykładów usunięto komentarze znajdujące się w kodzie oraz wywołania funkcji logujących informacje celem skrócenia listingów. Całość kodu aplikacji dostępna jest w załączonym do opracowania archiwum.

3.4.1 Aplikacja mobilna

Wysyłanie wiadomości zostało rozbite na kilka funkcji ze względu na złożoność tego procesu, oraz aby zachować odrębność odpowiedzialności między jednostkami kodu. Funkcja nadrzędna (Listing 3.1) deleguje przygotowanie wiadomości w formacie przewidzianym do wysyłki (zaszyfrowane algorytmem SDEx oraz uzupełnione o metadane) do funkcji `prepareToSend` (Listing 3.2). Sama natomiast odpowiada za pobranie wymaganych informacji (klucz nadawcy) z pamięci aplikacji oraz zarządzanie przebiegiem procesu w zależności od powodzenia lub niepowodzenia kolejnych jego etapów.

Samo wysyłanie wiadomości jest delegowane do funkcji `executeSendMessage` (Listing 3.3), która warunkując to powodzeniem dostarczenia wiadomości dodaje przesłaną wiadomość do lokalnej bazy danych, aby była widoczna w historii konwersacji. Jeśli natomiast wysyłka się nie powiedzie wiadomość nie jest zapisywana, dzięki temu obaj rozmówcy mają spójną historię rozmowy.

```
export async function sendMessage(
    message: Message,
    publicKeyTo: string,
    sqlDbSession: WebSQLDatabase,
    sdexEngine: SdexCrypto,
): Promise<boolean> {
    const publicKeyFrom = mmkvStorage.getString("publicKey");
    if (!publicKeyFrom) {
        throw new PreconditionError(
            "[sendMessage] First party's key(s) not found. Cannot send
               message.",
        );
    }
    const transportReadyMessage: TransportedMessage = await
       prepareToSend(
        message,
        publicKeyFrom,
        publicKeyTo,
        sdexEngine,
    );
    message=${JSON.stringify(transportReadyMessage)}`);
    const successfulSend = await executeSendMessage(
       transportReadyMessage, message, sqlDbSession);
    if (successfulSend) {
        return true;
    }
    return false;
}
```

Listing kodu 3.1: Wysyłanie wiadomości - funkcja nadrzędna

```
export function prepareToSend(
    message: Message,
    publicKeyFrom: string,
    publicKeyTo: string,
    sdexEngine: SdexCrypto,
): TransportedMessage {
    const sdexEncryptedText = bytesToBase64(sdexEngine.encryptMessage(
       message.text));
    const sdexEncryptedImage = message.image
        ? bytesToBase64(sdexEngine.encryptMessage(message.image))
        : undefined;
    const sdexEncryptedVideo = message.video
        ? bytesToBase64(sdexEngine.encryptMessage(message.video))
        : undefined;
    const sdexEncryptedAudio = message.audio
        ? bytesToBase64(sdexEngine.encryptMessage(message.audio))
        : undefined;

    const sdexEncryptedMessage = new Message(
        message.contactIdFrom,
        message.contactIdTo,
```

```
        sdexEncryptedText,
        message.createdAt,
        message.unread,
        sdexEncryptedImage,
        sdexEncryptedVideo,
        sdexEncryptedAudio,
        message.id,
    );

    const transportedMessage = messageToTransportedMessage(
        sdexEncryptedMessage,
        publicKeyFrom,
        publicKeyTo,
    );

    return transportedMessage;
}
```

Listing kodu 3.2: Przygotowywanie wiadomości do wysyłki

```
async function executeSendMessage(
    transportedMessage: TransportedMessage,
    message: Message,
    sqlDbSession: WebSQLDatabase,
): Promise<boolean> {
    try {
        const response = await socket.emitWithAck("chat",
            transportedMessage);
        if (response === "success") {
            try {
                await addMessage(message, sqlDbSession);
                return true;
            } catch (error) {
                return false;
            }
        } else {
            return false;
        }
    } catch (error) {
        return false;
    }
}
```

Listing kodu 3.3: Wysyłanie wiadomości - funkcja odpowiedzialna za komunikację z serwerem i dodanie wiadomości do bazy danych

Kolejnym znaczącym przykładem jest proces wymiany kluczy sesji między użytkownikami oraz odpowiednie przechowanie go w pamięci niezachowawczej aplikacji. Klient wychodzący z inicjatywą komunikacji (co jest wywołane poprzez otwarcie okna czatu z wybranym przez siebie kontaktem) wywołuje funkcję `initiateChat` (Listing 3.4). Uzyskany klucz sesji jest przechowywany w pamięci. Implementacja sposobu przechowywania tej informacji w tzw. stanie aplikacji z użyciem biblioteki Zustand jest pokazana na Listingu 3.5.

```
export async function initiateChat(publicKeyTo: string): Promise<boolean> {
    const existingSessionKey = useCryptoContextStore.getState().sdexEngines.has(publicKeyTo);
    if (existingSessionKey) {
        return true;
    }
    const firstPartyPublicKey = mmkvStorage.getString("publicKey");
    const firstPartyPrivateKey = mmkvStorage.getString("privateKey");

    if (!firstPartyPublicKey || !firstPartyPrivateKey) {
        throw new PreconditionError("First party's key(s) not found. Cannot initiate chat.");
    }

    const sessionKeyFirstPart = generateSessionKeyPart();
    const sessionKeyFirstPartString = bytesToBase64(sessionKeyFirstPart);

    const sessionKeyFirstPartEncrypted = await encryptRsa(publicKeyTo, sessionKeyFirstPartString);

    try {
        const response = await socket.emitWithAck("chatInit", {
            publicKeyFrom: firstPartyPublicKey,
            publicKeyTo,
            sessionKeyPartEncrypted: sessionKeyFirstPartEncrypted,
        });
        if (!response) {
            return false;
        }
        try {
            const sessionKeySecondPartString = await decryptRsa(firstPartyPrivateKey, response);
            const sessionKeySecondPart = base64ToBytes(sessionKeySecondPartString);
            const sessionKey = mergeUint8Arrays(sessionKeyFirstPart, sessionKeySecondPart);

            const sdexEngine = new SdexCrypto(sessionKey);
            useCryptoContextStore.getState().addUserEngine(publicKeyTo, sdexEngine);

            return true;
        } catch (error) {
            return false;
        }
    } catch (error) {
        return false;
    }
}
```

Listing kodu 3.4: Inicjowanie komunikacji z innym użytkownikiem

```
export const useCryptoContextStore = create<CryptoContextState>((set) => ({
    sdexEngines: new Map<string, SdexCrypto>(),
    addUserEngine: (publicKey: string, sdexEngine: SdexCrypto): void => {
        set((prev) => ({
            sdexEngines: new Map(prev.sdexEngines).set(publicKey, sdexEngine),
        }));
    },
}));
```

Listing kodu 3.5: Definicja jednego ze stanów aplikacji przechowującego informacje o kluczach sesji i odpowiadających im kontaktach

Proces szyfrowania i deszyfrowania metodą SDEx opisany w sekcji 2.1 został zaimplementowany w sposób pokazany na Listingu 3.6

```
export default class SdexCrypto {
    HASH_LENGTH: number;
    ZEROED_BLOCK: Uint8Array;
    sessionKeyHash: Uint8Array;
    sessionKeyFirstPartHash: Uint8Array; // aka S1
    sessionKeySecondPartHash: Uint8Array; // aka S2

    constructor(sessionKey: Uint8Array, hashLength = 32) {
        this.HASH_LENGTH = hashLength;
        this.ZEROED_BLOCK = new Uint8Array(this.HASH_LENGTH);
        this.sessionKeyHash = this.blake3Wrapper(sessionKey);
        const [sessionKeyFirstPart, sessionKeySecondPart] = splitSessionKey(sessionKey);
        this.sessionKeyFirstPartHash = this.blake3Wrapper(sessionKeyFirstPart as Uint8Array);
        this.sessionKeySecondPartHash = this.blake3Wrapper(sessionKeySecondPart as Uint8Array);
    }

    blake3Wrapper(message: Uint8Array | string, context?: Uint8Array | string) {
        return blake3(message, {
            dkLen: this.HASH_LENGTH,
            context,
        });
    }

    static calculateBlock(block: Uint8Array, hash1: Uint8Array, hash2: Uint8Array): Uint8Array {
        if (block.length !== hash1.length || block.length !== hash2.length) {
            throw new SdexEncryptionError("[SdexCrypto.calculateBlock] Invalid block length");
        }
        return xorUintArrays(block, hash1, hash2);
    }

    calculateMessage(messageByteArray: Uint8Array): Uint8Array {
        const messageSplit = splitMessageIntoBlocks(messageByteArray, this.HASH_LENGTH);
```

```
        const messageBlocks = changeTo1IndexedArray(messageSplit);
        const result = <Uint8Array[]>[];
        const hashIterations = <Uint8Array[]>[]; // h0, h1, ..., hk

        result[1] = SdexCrypto.calculateBlock(
            messageBlocks[1] as Uint8Array,
            this.sessionKeyFirstPartHash,
            this.sessionKeyHash,
        );
        hashIterations[1] = this.blake3Wrapper(
            this.sessionKeyHash,
            mergeUint8Arrays(messageBlocks[1] as Uint8Array,
               messageBlocks[2] ?? this.ZEROED_BLOCK),
        );

        if (messageBlocks[2]) {
            result[2] = SdexCrypto.calculateBlock(
                messageBlocks[2],
                this.sessionKeyFirstPartHash,
                this.sessionKeySecondPartHash,
            );
            hashIterations[2] = this.blake3Wrapper(
                xorUintArrays(hashIterations[1], this.sessionKeyHash),
                mergeUint8Arrays(
                    messageBlocks[3] ?? this.ZEROED_BLOCK,
                    messageBlocks[4] ?? this.ZEROED_BLOCK,
                ),
            );
        }
        for (let k = 1; k <= messageBlocks.length; k += 1) {
            if (k >= 3) {
                hashIterations[k] = this.blake3Wrapper(
                    xorUintArrays(
                        hashIterations[k - 1] ?? this.ZEROED_BLOCK,
                        hashIterations[k - 2] ?? this.ZEROED_BLOCK,
                    ),
                    mergeUint8Arrays(
                        messageBlocks[2 * k - 1] ?? this.ZEROED_BLOCK,
                        messageBlocks[2 * k] ?? this.ZEROED_BLOCK,
                    ),
                );
                result[2 * k + 1] = SdexCrypto.calculateBlock(
                    messageBlocks[2 * k + 1] ?? this.ZEROED_BLOCK,
                    hashIterations[k] ?? this.ZEROED_BLOCK,
                    hashIterations[k - 1] ?? this.ZEROED_BLOCK,
                );
                result[2 * k + 2] = SdexCrypto.calculateBlock(
                    messageBlocks[2 * k] ?? this.ZEROED_BLOCK,
                    this.sessionKeySecondPartHash,
                    hashIterations[k] ?? this.ZEROED_BLOCK,
                );
            }
        }
        const nonEmptyArray = result.filter((element) => element);

        return mergeUint8Arrays(...nonEmptyArray);
    }

```

```
90      encryptMessage(message: string): Uint8Array {
91          const messageByteArray = stringToBytes(message);
92          const encrypted = this.calculateMessage(messageByteArray);
93          return encrypted;
94      }
95
96      decryptMessage(messageCipherTextByteArray: Uint8Array): string {
97          const decryptedByteArray = this.calculateMessage(
                messageCipherTextByteArray);
98          for (let i = decryptedByteArray.length - 1; i >= 0; i -= 1) {
99              if (decryptedByteArray[i] !== 0) {
100                 const decrypted = bytesToString(decryptedByteArray.slice
                        (0, i + 1));
101                 logger.debug(`[SdexCrypto.decryptMessage] (SDEx)
                        Decrypted message: ${decrypted}`);
102                 return decrypted;
103             }
104         }
105         return "";
106     }
107 }
```

Listing kodu 3.6: Szyfrowanie i deszyfrowanie metodą SDEx - implementacja

Najbardziej rozbudowanym pod względem funkcjonalności i zależności od różnych modułów aplikacji jest widok czatu, korzysta on zarówno w tle jak i w reakcji na działania użytkownika z szeregu funkcji komunikujących się z serwerem (np. aby sprawdzić status połączenia adresata i nadawcy z serwerem). Implementację tego widoku przedstawia Listing 3.7.

Aby zapewnić odpowiednią responsywność aplikacji na przychodzące wiadomości (co jest szczególnie istotne gdy użytkownik jest w oknie czatu) podmieniono funkcję nasłuchującą wiadomości w tle na jej odpowiednik wywoływany bezpośrednio w komponencie definiującym widok czatu, dzięki temu w ramach tego widoku można natychmiast reagować na przychodzące tym kanałem wiadomości i odświeżać widok, aby wyświetlić otrzymaną wiadomość.

```
1  export default function Chat() {
2      const [messages, setMessages] = React.useState<GiftedChatMessage
           []>([]);
3      const [contact, setContact] = React.useState<Contact | undefined>(
           undefined);
4      const [firstPartyContact, setFirstPartyContact] = React.useState<
           Contact | undefined>(
5          undefined,
6      );
7      const [bannerOfflineVisible, setBannerOfflineVisible] = React.
           useState<boolean>(false);
8      const [thirdPartyOnline, setThirdPartyOnline] = React.useState<
           boolean>(true);
9      const [thirdPartyKeyRegistered, setThirdPartyKeyRegistered] = React.
           useState<boolean>(true);
10     const sqlDbSession = useSqlDbSessionStore((state) => state.
           sqlDbSession);
11     const sdexEngines = useCryptoContextStore((state) => state.
           sdexEngines);
12     const params = useLocalSearchParams();
13     const contactId = params.contactId !== undefined ? Number(params.
```

```
        contactId) : undefined;

    React.useEffect(() => {
        requestRegister();
    }, []);

    React.useEffect(() => {
        socket.off("chat", outsideChatRoomChatListener);
        socket.on(
            "chat",
            async (
                message: TransportedMessage,
                callback: (response: boolean) => void,
            ): Promise<void> => {
                const sdexEngine = useCryptoContextStore
                    .getState()
                    .sdexEngines.get(message.publicKeyFrom);
                if (!sdexEngine) {
                    callback(false);
                    return;
                }
                const firstPartyPrivateKey = mmkvStorage.getString("
                    privateKey");
                if (!firstPartyPrivateKey) {
                    callback(false);
                    return;
                }
                if (!sqlDbSession) {
                    callback(false);
                    return;
                }
                const contactFrom = await getContactByPublicKey(
                    message.publicKeyFrom,
                    sqlDbSession,
                );
                if (!contactFrom) {
                    callback(false);
                    return;
                }
                const decryptedMessage = await prepareToIngest(
                    message,
                    sdexEngine,
                    firstPartyPrivateKey,
                    contactFrom.id as number, // if it's fetched from db
                        we know it has an id
                );
                await addMessage(decryptedMessage, sqlDbSession);

                if (firstPartyContact && contact?.id ===
                    decryptedMessage.contactIdFrom) {
                    const giftedChatMessage = messageToGiftedChatMessage
                        (
                        decryptedMessage,
                        contact,
                        firstPartyContact,
                    );
                    setMessages([giftedChatMessage, ...messages]);
                }
```

```
67                  callback(true);
68              },
69          );
70          return () => {
71              socket.on("chat", outsideChatRoomChatListener);
72          };
73      }, []);
74
75      React.useEffect(() => {
76          if (contactId) {
77              (async () => {
78                  const fetchedContact = await getContactById(Number(
                        contactId), sqlDbSession);
79                  if (!fetchedContact) {
80                      logger.error(
81                          `[Chat.useEffect] User with contactId=${JSON.
                                stringify(
82                                 contactId,
83                              )} not found in storage.`,
84                      );
85                  } else {
86                      setContact(fetchedContact);
87                  }
88                  const fetchedFirstPartyContact = await getContactById(0,
                        sqlDbSession);
89                  if (!fetchedFirstPartyContact) {
90                      logger.error(
91                          `[Chat.useEffect] First party contact info not
                                found in the database..`,
92                      );
93                  } else {
94                      setFirstPartyContact(fetchedFirstPartyContact);
95                  }
96              })();
97          }
98      }, [contactId]);
99
100     function runOnlineCheck(publicKey: string) {
101         checkOnline(publicKey)
102             .then((online) => {
103                 logger.info(
104                     `[Chat.useEffect] Third party connected is to the
                            server: ${JSON.stringify(
105                             online,
106                         )}.`,
107                 );
108                 setThirdPartyOnline(online);
109             })
110             .catch((error: Error) => {
111                 logger.error(
112                     `[Chat.useEffect] Error while checking if third
                            party is online: ${error.message}`,
113                 );
114             });
115     }
116
117     React.useEffect(() => {
118         if (contact?.publicKey) {
```

```
119             runOnlineCheck(contact.publicKey);
120             const interval = setInterval(() => {
121                 runOnlineCheck(contact.publicKey);
122             }, 10000);
123             return () => clearInterval(interval);
124         }
125     }, [contact?.publicKey]);
126
127     React.useEffect(() => {
128         if (contact?.publicKey) {
129             checkKey(contact.publicKey)
130                 .then((registered) => {
131                     setThirdPartyKeyRegistered(registered);
132                 })
133                 .catch((error: Error) => {
134                     logger.error(
135                         `[Chat.useEffect] Error while checking if key is
                                registered: ${error.message}`,
136                     );
137                 });
138         }
139     }, [contact?.publicKey]);
140
141     React.useEffect(() => {
142         if (contact?.publicKey && sdexEngines.has(contact?.publicKey)) {
143         } else if (
144             socket.connected &&
145             contact?.publicKey &&
146             thirdPartyOnline &&
147             !sdexEngines.has(contact.publicKey)
148         ) {
149             logger.info(`[Chat.useEffect] Sending "chatInit" message to
                ${contact.getFullName()}.`);
150             void initiateChat(contact.publicKey);
151         }
152     }, [socket.connected, contact?.publicKey, thirdPartyOnline]);
153
154     React.useEffect(() => {
155         if (contact && contact.id && firstPartyContact) {
156             (async () => {
157                 const messagesFromStorage = await getMessagesByContactId
                    (
158                     Number(contactId),
159                     sqlDbSession,
160                 );
161                 await markMessagesAsRead(contact.id as number,
                    sqlDbSession);
162                 const giftedChatMessages: GiftedChatMessage[] = [];
163                 messagesFromStorage.forEach((message) => {
164                     giftedChatMessages.push(
165                         messageToGiftedChatMessage(message, contact,
                            firstPartyContact),
166                     );
167                 });
168                 setMessages(giftedChatMessages);
169             })();
170         }
171     }, [contact, firstPartyContact]);
```

```
172
173     React.useEffect(() => {
174         if (!socket.connected || !thirdPartyOnline || !
                thirdPartyKeyRegistered) {
175             setBannerOfflineVisible(true);
176         } else {
177             setBannerOfflineVisible(false);
178         }
179     }, [thirdPartyOnline, socket.connected, thirdPartyKeyRegistered]);
180
181     async function onSend(newMessages: GiftedChatMessage[] = []):
            Promise<void> {
182         setMessages((previousMessages) => GiftedChat.append(
                previousMessages, newMessages));
183         const thirdPartyCryptoEngine = sdexEngines.get(contact?.
                publicKey as string);
184         if (
185             contact &&
186             socket.connected &&
187             sqlDbSession &&
188             thirdPartyCryptoEngine &&
189             thirdPartyOnline
190         ) {
191             const results = [];
192             for (const msg of newMessages) {
193                 const message = giftedChatMessageToMessage(msg, contact.
                        id as number);
194                 results.push(
195                     sendMessage(message, contact.publicKey, sqlDbSession
                            , thirdPartyCryptoEngine),
196                 );
197             }
198             await Promise.all(results);
199         }
200     }
201
202     function determineBannerText(): string {
203         let text = "";
204         if (!socket.connected) {
205             text =
206                 "Brak połączenia z serwerem. Aby wysyła  i odbiera
                        wiadomości, połącz się z serwerem.";
207         } else if (!thirdPartyKeyRegistered) {
208             text =
209                 "Klucz publiczny użytkownika nie jest zarejestrowany na
                        serwerze (prawdopodobnie jest nieprawidłowy).";
210         } else if (!thirdPartyOnline) {
211             text =
212                 "Użytkownik jest offline. Możesz wysyła  wiadomości,
                        ale nie ma gwarancji, że dotrą.";
213         }
214         return text;
215     }
216
217     function determineBannerActionLabel(): string {
218         let text = "";
219         if (!socket.connected) {
220             text = "Polacz";
```

```
221         } else if (!thirdPartyOnline || !thirdPartyKeyRegistered) {
222             text = "Okej";
223         }
224         return text;
225     }
226 
227     function determineBannerAction(): void {
228         let action = () => {};
229         if (!socket.connected) {
230             action = socketConnect;
231         } else if (!thirdPartyOnline) {
232             action = () => setBannerOfflineVisible(false);
233         }
234         return action();
235     }
236 
237     return (
238         <SafeAreaView className="flex-1">
239             <Appbar.Header style={styles.appBarHeader}>
240                 <Appbar.Content
241                     title={contact ? contact.getFullName() : "<name>"}
242                     titleStyle={styles.appBarTitle}
243                 />
244                 <Link href="/chats" asChild>
245                     <Appbar.BackAction iconColor={styles.appBarIcons.
                            color} />
246                 </Link>
247             </Appbar.Header>
248             <Banner
249                 visible={bannerOfflineVisible}
250                 icon="connection"
251                 actions={[
252                     {
253                         label: determineBannerActionLabel(),
254                         onPress: determineBannerAction,
255                     },
256                 ]}
257             >
258                 {determineBannerText()}
259             </Banner>
260             <GiftedChat
261                 messages={messages}
262                 textInputProps={{ autoFocus: true }}
263                 onSend={(newMessages) => onSend(newMessages)}
264                 user={{
265                     _id: 0,
266                     name: firstPartyContact?.getFullName(),
267                 }}
268             />
269         </SafeAreaView>
270     );
271 }
```

Listing kodu 3.7: Implementacja ekranu czatu

3.4.2 Aplikacja serwerowa

Większość funkcji obsługujących odbiór wiadomości od urządzenia klienckiego weryfikuje dane przychodzące. Zastosowano prostą logikę weryfikującą kompletność oraz typ wymaganych parametrów. Listing kodu 3.8 pokazuje typową funkcję weryfikującą dane przychodzące dla wydarzenia finalizującego proces rejestracji na serwerze.

```
def validate_register_follow_up_payload(data: Any) -> bool:
    if not isinstance(data, dict):
        return False
    if not data.get("login", None):
        return False
    if not data.get("publicKey", None):
        return False
    if not data.get("signature", None):
        return False
    return True
```

Listing kodu 3.8: Walidacja danych przychodzących

Endpoint obsługujący przekazywanie wiadomości między użytkownikami (Listing 3.9) jest przykładem ilustrującym implementację autoryzacji użytkownika do wykonywania pewnych akcji, a także pokazuje logikę obsługi trójstronnej komunikacji (klient 1 - serwer - klient 2).

```
@socket_manager.on("chat")  # type: ignore
async def handle_chat(sender_sid: str, data: Any) -> ResponseStatusType:
    if not validate_chat_payload(data):
        return "error"

    sender_key = PUBLIC_KEYS_SIDS_MAPPING.inverse.get(sender_sid, None)
    receiver_sid = PUBLIC_KEYS_SIDS_MAPPING.get(data["publicKeyTo"],
                                                None)
    if not sender_key or not receiver_sid:
        return "error"

    if not is_authenticated(sender_sid):
        return "error"
    if not is_authenticated(receiver_sid):
        return "error"

    try:
        receiver_response: bool = await socket_manager.call(
            "chat",
            data,
            to=receiver_sid,
        )
        if receiver_response:
            return "success"
        else:
            return "error"
    except TimeoutError:
        return "error"
```

Listing kodu 3.9: Przekazywanie wiadomości między użytkownikami

Operacje wykonywane na bazie danych (SQLite) pokazuje Listing 3.10.

```
class DatabaseManager:
    def __init__(self, db_path: Path | str) -> None:
        try:
            self.client: sqlite3.Connection = sqlite3.connect(db_path)
        except Exception as e:
            raise DBConnectionError(e)

    def get_user_by_login(self, login: str) -> User | None:
        try:
            cursor: sqlite3.Cursor = self.client.execute(
                """
                SELECT
                    id, login, public_key
                FROM
                    users
                WHERE
                    login = :login;
                """,
                {"login": login},
            )
            output = cursor.fetchone()
            if not output:
                logger.info("User not found.")
                return None
            user = User(
                id=output[0],
                login=output[1],
                public_key=output[2],
            )
            logger.info("User data fetched successfully.")
            return user
        except Exception as e:
            raise DBConnectionError(e)

    def check_public_key(self, public_key: str) -> bool:
        try:
            cursor: sqlite3.Cursor = self.client.execute(
                """
                SELECT
                    *
                FROM
                    users
                WHERE
                    public_key = :key;
                """,
                {"key": public_key},
            )
            output = cursor.fetchone()
            log_msg = "Public key exists." if output else "Public key
                                                    does not exist."
            logger.info(log_msg)
            return True if output else False
        except Exception as e:
            raise DBConnectionError(e)

    def update_user(self, login: str, new_public_key: str) -> bool:
        try:
```

```
57              self.client.execute(
58                  """
59                  UPDATE
60                      users
61                  SET
62                      public_key = :new_public_key
63                  WHERE
64                      login = :login;
65                  """,
66                  {
67                      "new_public_key": new_public_key,
68                      "login": login,
69                  },
70              )
71              self.client.commit()
72              return self.client.total_changes > 0
73          except Exception as e:
74              raise DBConnectionError(e)
75
76      def add_user(self, user: User) -> bool:
77          try:
78              self.client.execute(
79                  """
80                  INSERT INTO users (login, public_key)
81                  VALUES (:login, :public_rsa);
82                  """,
83                  {"login": user.login, "public_rsa": user.public_key},
84              )
85              self.client.commit()
86              return self.client.total_changes > 0
87          except Exception as e:
88              raise DBConnectionError(e)
89
90      def remove_user(self, login: str) -> bool:
91          try:
92              self.client.execute(
93                  """
94                  DELETE FROM
95                      users
96                  WHERE
97                      login = :login;
98                  """,
99                  {"login": login},
100             )
101             self.client.commit()
102             return self.client.total_changes > 0
103         except Exception as e:
104             raise DBConnectionError(e)
```

Listing kodu 3.10: Implementacja menadżera bazy danych

3.5 Testy

Obie aplikacje zostały poddane testom mającym na celu usprawnienie pracy nad rozwijaniem omawianego oprogramowania oraz weryfikację poprawności stworzonego rozwiązania również w trakcie dalszej pracy nad kodem polegającej na jego optymalizacji i

poprawy jakości (refactoringu).

Napisane zostały testy jednostkowe oraz integracyjne. Testowanie działania całości opisanych funkcjonalności (tzw. testy *end to end*) oraz poprawności wyświetlania interfejsów graficznych przeprowadzono manualnie z użyciem emulatorów i fizycznych urządzeń z systemem Android.

Wybrane testy przedstawiają Listingi 3.11-3.13.

```
1  const sessionKey = new Uint8Array([
2      199, 182, 158, 16, 28, 191, 237, 76, 143, 157, 160, 176, 212, 216,
           69, 149, 116, 80, 98, 155,
3      212, 183, 228, 53, 100, 16, 112, 89, 150, 82, 0, 116, 163, 242, 21,
           164, 67, 83, 188, 5, 92, 26,
4      189, 251, 17, 55, 89, 90, 4, 193, 80, 49, 150, 142, 205, 68, 98, 31,
            22, 221, 192, 211, 235, 55,
5  ]);
6  const clearTextMessage = "Hello world!";
7  const encryptedMessage = new Uint8Array([
8      216, 52, 125, 5, 37, 138, 143, 114, 25, 15, 52, 201, 212, 18, 223,
           193, 158, 24, 12, 232, 141,
9      40, 144, 183, 142, 15, 134, 10, 228, 223, 72, 148,
10 ]);
11 const sdexEngine = new SdexCrypto(sessionKey, 32);
12
13 test("Generating a session key part.", () => {
14     const sessionKeyPart = generateSessionKeyPart(32);
15     expect(sessionKeyPart.length).toBe(32);
16 });
17
18 test("Calculating a block properly.", () => {
19     const array1 = new Uint8Array([121, 133]); //   01111001 10000101
20     const array2 = new Uint8Array([38, 6]); //      00100110 00000110
21     const array3 = new Uint8Array([130, 69]); //    10000010 01000101
22     const result = SdexCrypto.calculateBlock(array1, array2, array3);
23     const expected = new Uint8Array([221, 198]); // 11011101 11000110
24     expect(result).toEqual(expected);
25 });
26
27 test("Calculating a block throws error on mismatching arrays lengths.",
       () => {
28     const array1 = new Uint8Array([121, 133]);
29     const array2 = new Uint8Array([9]);
30     const array3 = new Uint8Array([130, 69]);
31     const result = () => {
32         SdexCrypto.calculateBlock(array1, array2, array3);
33     };
34     expect(result).toThrow(SdexEncryptionError);
35     expect(result).toThrow("Invalid block length");
36 });
37
38 test("Encrypting a message properly.", () => {
39     const result = sdexEngine.encryptMessage(clearTextMessage);
40     expect(result).toEqual(encryptedMessage);
41 });
42
43 test("Decrypting a message properly.", () => {
44     const result = sdexEngine.decryptMessage(encryptedMessage);
45     expect(result).toBe(clearTextMessage);
46 });
```

```
47
48 test("SDEx encryption is reversible.", () => {
49     const encrypted = sdexEngine.encryptMessage(clearTextMessage);
50     const decrypted = sdexEngine.decryptMessage(encrypted);
51     expect(clearTextMessage).toBe(decrypted);
52 });
```

Listing kodu 3.11: Testy jednostkowe sprawdzające implementację algorytmu SDEx

```
1  test("Preparation process for message is fully reversible by ingestion
       process", async () => {
2      const originalMessage = new Message(0, 1, "this is a test.", new
           Date("2023-01-02"), false);
3      const senderPublicKey = `-----BEGIN PUBLIC KEY-----
4    MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCzc5oViIvknxXSbuIfqkyaZc1F
5    nVvN52Buu136pSC6AbfVGX5KHzR7lzJl1ESxarREY8rrb8QxsNs+FAntuwiZopdW
6    8f4zHB91neApkSLtuos4k6Gu78KvbldHkeCx8BdQsWz03lNXpv5REp9wNKGyzenw
7    wF1dAlLOSg60efyUkwIDAQAB
8    -----END PUBLIC KEY-----`;
9      const receiverPublicKey = `-----BEGIN PUBLIC KEY-----
10   MIGeMA0GCSqGSIb3DQEBAQUAA4GMADCBiAKBgFn8Dq9VxuIjiBzLmLZY5HUkHcr7
11   czMdBKmsY3CiK6zauqmIXZqYLadVJBTh2+v2/kShiQViY+i9HbTJbzl7BTw6p8fr
12   B0NWaNdlul/i0EkQZs45dJHe2HZebw7ZbOJmPyhUZAzmzastB7u69qNJANfxIFVB
13   uII/u4ssaGki5iTJAgMBAAE=
14   -----END PUBLIC KEY-----`;
15     // for this test we skip combining session key parts and just
           provide a ready full length session key
16     const senderSessionKey = generateSessionKeyPart(64);
17     const sdexEngine = new SdexCrypto(senderSessionKey);
18
19     const messageToSend = prepareToSend(
20         originalMessage,
21         senderPublicKey,
22         receiverPublicKey,
23         sdexEngine,
24     );
25     const ingestedMessage = prepareToIngest(messageToSend, sdexEngine,
           1);
26     expect(ingestedMessage.contactIdFrom).toEqual(1);
27     expect(ingestedMessage.contactIdTo).toEqual(0);
28     expect(ingestedMessage.text).toEqual(originalMessage.text);
29     expect(ingestedMessage.createdAt).toEqual(originalMessage.createdAt)
           ;
30     expect(ingestedMessage.image).toEqual(originalMessage.image);
31     expect(ingestedMessage.video).toEqual(originalMessage.video);
32     expect(ingestedMessage.audio).toEqual(originalMessage.audio);
33 });
```

Listing kodu 3.12: Test integracyjny sprawdzający poprawność przygotowywania wiadomości do wysyłki, a następnie do jej zapisania u odbiorcy

```python
import pathlib

import pytest

from sdex_server.database.database import DatabaseManager
from sdex_server.database.models import User


@pytest.fixture
def db_manager() -> DatabaseManager:
    db_path = pathlib.Path(__file__).parent.parent.parent / "resources" / "test-db.db"
    db_manager = DatabaseManager(db_path)
    return db_manager


@pytest.fixture
def user() -> User:
    return User(id=123, public_key="rsa-test", login="test_login")


@pytest.fixture
def updating_user(db_manager: DatabaseManager) -> bool:
    previous_user = User(public_key="old-rsa", login="some_user")
    new_user = User(public_key="new-rsa", login="some_user")
    yield db_manager.update_user(new_user.login, new_user.public_key)  # type: ignore
    db_manager.update_user(previous_user.login, previous_user.public_key)


@pytest.fixture
def adding_user(db_manager: DatabaseManager) -> bool:
    user_to_add = User(public_key="added-user-key", login="added-user")
    yield db_manager.add_user(user_to_add)  # type: ignore
    db_manager.remove_user(user_to_add.login)


@pytest.fixture
def deleting_user(db_manager: DatabaseManager) -> bool:
    user_to_delete = User(public_key="test-key", login="test123")
    yield db_manager.remove_user(user_to_delete.login)  # type: ignore
    db_manager.add_user(user_to_delete)


def test_get_user_by_login_returns_user(
    db_manager: DatabaseManager, user: User
) -> None:
    assert db_manager.get_user_by_login(user.login) == user


def test_get_user_by_login_doesnt_find_user(db_manager: DatabaseManager) -> None:
    assert db_manager.get_user_by_login("some_false_login") is None


def test_check_public_key_public_returns_public_key_exists(
```

```
54      db_manager: DatabaseManager,
55  ) -> None:
56      assert db_manager.check_public_key("rsa-test") is True
57
58
59  def test_check_public_key_public_returns_public_key_doesnt_exist(
60      db_manager: DatabaseManager,
61  ) -> None:
62      assert db_manager.check_public_key("false-rsa") is False
63
64
65  def test_update_user_executes_successfully(updating_user: bool) -> None:
66      assert updating_user is True
67
68
69  def test_update_user_public_key_doesnt_find_user(
70      db_manager: DatabaseManager, user: User
71  ) -> None:
72      assert db_manager.update_user("doesnt-exist", "some-key") is False
73
74
75  def test_add_user_inserts_user_successfully(adding_user: bool) -> None:
76      assert adding_user is True
77
78
79  def test_remove_user_deletes_user_successfully(deleting_user: bool) ->
                                                   None:
80      assert deleting_user is True
```

Listing kodu 3.13: Testy jednostkowe sprawdzające operacje wykonywane na bazie danych

Rozdział 4

Dokumentacja deweloperska aplikacji

4.1 Stos technologiczny

W sekcji 2.5 opisane i uargumentowane zostały główne wybory technologii do stworzenia aplikacji dla części praktycznej niniejszego opracowania. W tej części uszczegółowiono wybór istotniejszych (według kryterium funkcjonalności) rozwiązań wraz z krótką charakterystyką, za co są odpowiedzialne w stworzonej aplikacji.

4.1.1 Aplikacja mobilna

- react-native – framework odpowiedzialny za tworzenie komponentów aplikacji takich jak poszczególne widoki (widok okna czatu, lista czatów (*chat rooms*, widok ustawień), sam framework umożliwia lub ułatwia też korzystanie z szeregu bibliotek rozwijanych przez Meta Open Source[11] oraz społeczność niezależnych twórców.

- react-native-async-storage – biblioteka daje dostęp do pamięci zachowawczej aplikacji, dzięki czemu możliwe jest przechowywanie konfiguracji aplikacji i odtwarzanie jej po jej uruchomieniu. Pamięć ta jest nieszyfrowana, dlatego dane wrażliwe, takie jak klucze RSA użytkownika i hasło logowania przechowywane są w inny sposób.

- react-native-gifted-chat – biblioteka implementująca widok i funkcjonalności czatu, dzięki której dodając własną logikę odbierania wiadomości z serwera czy wczytywania archiwalnych wiadomości z pamięci można w prosty sposób wyświetlać otrzymane i wysłane wiadomości oraz tworzyć nowe.

- react-native-mmkv – biblioteka umożliwiająca korzystanie z bardzo szybkiej bazy danych MMKV typu klucz-wartość na urządzenia mobilne. Baza danych daje możliwość szyfrowania jej zawartości, dlatego została wykorzystana do przechowywania poufnych danych użytkownika, takich jak login i hasło logowania.

- react-native-qrcode-svg – biblioteka do generowania kodów QR wykorzystana do tworzenia obrazków z kodami zawierającymi klucz publiczny użytkownika, które można przesyłać osobom, z którymi chce się komunikować.

- react-native-rsa-native – biblioteka implementująca generowanie kluczy RSA i funkcjonalności z nimi związane (szyfrowanie, deszyfrowanie, podpisywanie dokumentów). Biblioteka została zaimplementowana w sposób natywny (z użyciem języków

Java, Swift i Objective C) i jest bardzo wydajna, co przekłada się na krótki czas generowania kluczy.

- react-navigation – biblioteka udostępniająca komponenty umożliwiające użytkownikowi nawigację, czyli przechodzenia między widokami. Są to elementy takie jak dolny pasek nawigacji z zakładkami.

- expo – jest to platforma, w skład której wchodzi zestaw bibliotek ułatwiających tworzenie aplikacji w oparciu o framework React Native. Sama platforma zapewnia też narzędzia, zarówno lokalne (expo-cli) jak i chmurowe (*Expo Application Services*, EAS) wspomagające proces budowania i debugowania aplikacji, publikowania aplikacji do sklepów Apple App Store i Google Play Store oraz zarządzania dependencjami czy migracji do kolejnych wersji SDK platformy. W skład tej platformy wchodzą liczne biblioteki ułatwiające pracę z różnymi komponentami systemu operacyjnego urządzenia mobilnego (np. zarządzanie uprawnieniami, dostęp do kamery, dostęp do systemu plików).

- expo-crypto – biblioteka oferująca różne funkcje kryptograficzne, wykorzystana została do generowania kryptograficznie bezpiecznych (o wysokim poziomie pseudo losowości) ciągów bitów wykorzystywanych jako klucze sesji.

- expo-file-system – biblioteka zapewniająca dostęp do systemu plików urządzenia. Istotnym ograniczeniem jest tu dostęp jedynie do plików w ramach pamięci przydzielonej danej aplikacji, nie ma możliwości uzyskania dostępu do plików użytkownika znajdujących się poza nią. Jednak na potrzeby stworzonej aplikacji było to wystarczające. Biblioteka została wykorzystana do zapisu i odczytu plików tekstowych oraz kodów QR zawierających klucze RSA.

- expo-media-library – biblioteka zapewniająca dostęp do albumów zdjęć i tzw. rolki aparatu w telefonie. Została wykorzystana do zapisania w nich wygenerowanych kodów QR w formie obrazków rastrowych.

- expo-barcode-scanner – biblioteka implementująca funkcjonalność skanera kodów QR, dzięki niej deweloper w prosty sposób może uruchamiać aparat urządzenia z odpowiednią konfiguracją oraz zarządzać rezultatami skanowania.

- expo-permissions – biblioteka pozwalająca w prosty sposób zarządzać uprawnieniami aplikacji do dostępu do pewnych funkcji i zasobów urządzenia, takich jak dostęp do systemu plików, lokalizacji czy dostęp do aparatu.

- expo-sqlite – biblioteka pozwalająca na dostęp do bazy danych SQLite. Baza ta jest wykorzystywana na urządzeniu do przechowywania historii wiadomości oraz danych o kontaktach użytkownika.

- expo-sharing – biblioteka umożliwiająca użytkownikowi przesłanie treści różnymi kanałami komunikacji. Została wykorzystana, aby dać klientowi możliwość przesłania kodu QR lub pliku tekstowego z kluczem publicznym. Użytkownik w interfejsie graficznym sam wybiera poprzez którą z dostępnych na jego urządzeniu aplikacji udostępnić te treści.

- noble-hashes - biblioteka implementująca wiele popularnych funkcji skrótu, wykorzystano zawartą w niej implementację funkcji hashującej BLAKE3.

- zustand – biblioteka do zarządzania stanem aplikacji (pamięcią podręczną). Wykorzystana została do przechowywania konfiguracji niezachowawczej programu oraz usprawnień podnoszących wydajność aplikacji, takich jak przechowywanie kontekstu potrzebnego do pewnych działań (np. klucze sesji wykorzystywane każdorazowo przy szyfrowaniu wysyłanych lub odszyfrowywaniu metodą SDEx otrzymywanych wiadomości).

- socket.io-client – biblioteka implementująca protokół WebSocket. Umożliwia ona m.in. nawiązywanie połączenia z serwerem, definiowanie emiterów wiadomości do serwera oraz modułów odpowiedzialnych za odbieranie wiadomości z niego przychodzących.

4.1.2 Aplikacja serwerowa

- fastapi – asynchroniczny framework webowy wykorzystany do stworzenia API umożliwiającego funkcjonowanie aplikacji mobilnej.

- fastapi-socketio – biblioteka służąca do integracji socket.io (implementacji protokołu WebSocket) z frameworkiem FastAPI

- uvicorn – implementacja serwera webowego w Pythonie. Wspiera HTTP/1.1, HTTP/2 i WebSocket.

- pydantic – biblioteka służąca do weryfikowania poprawności danych w oparciu o typowanie w Pythonie (tzw. *type hints*). W oparciu o tę bibliotekę zbudowany został także framework FastAPI. Biblioteka wykorzystywana jest do modelowania struktur danych znajdujących się w bazie danych serwera.

- rsa – biblioteka implementująca RSA w języku Python. Oferuje możliwość generowania pary kluczy, szyfrowania, deszyfrowania, a także podpisywania i weryfikacji podpisów.

4.2 Instalacja i uruchomienie aplikacji

4.2.1 Instrukcje dla aplikacji serwerowej

Do obsługi i utrzymania aplikacji serwerowej przygotowany został plik Makefile zawierający przygotowane komendy (tzw. *targets*) do uruchomienia aplikacji, testów, lintowania i formatowania kodu oraz uaktualniania bibliotek do nowszych wersji zgodnych z użytą wersją języka Python oraz zgodnych między sobą (rozwiązaniem kwestii zgodności, czyli zarządzaniem dependencjami zajmuje się narzędzie Poetry [5]). Wykorzystanie Makefile wiąże się z koniecznością posiadania zainstalowanego programu make (dostępnego na większości popularnych systemów operacyjnych), można jednak wykonać wyżej wymienione czynności bez niego kopiując komendy przypisane do wybranego targetu. Przykładowo aby zainstalować i uruchomić aplikację należy wywołać w wybranej przez siebie powłoce (będąc w katalogu głównym aplikacji serwerowej `kod_aplikacji/SDEx-encrypted-communicator/server`):

```
$ make install
$ make run
```

lub

```
$ poetry install
$ poetry run python sdex_server/main.py
```

pełna lista komend znajduje się w pliku:
`kod_aplikacji/SDEx-encrypted-communicator/server/Makefile`

4.2.2 Instrukcje dla aplikacji mobilnej

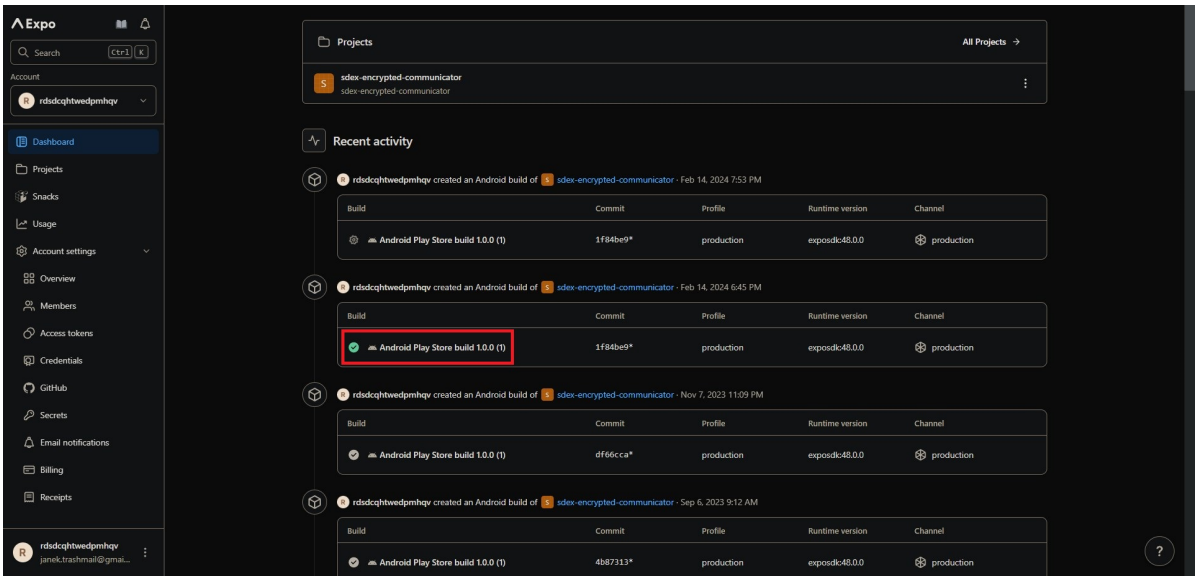
Aplikację mobilną można zainstalować wgrywając plik `.apk` znajdujący się w archiwum dołączonym do niniejszego opracowania.

Aby zbudować ten plik na bazie napisanego kodu należy przygotować środowisko (zainstalować npm, Expo, założyć konto na `https://expo.dev/` oraz zalogować się w powłoce przy pomocy swoich danych do konta komendą `eas login`, a następnie przejść do katalogu głównego aplikacji mobilnej:

`kod_aplikacji/SDEx-encrypted-communicator/server/sdex-encrypted-communicator` i wywołać komendę:

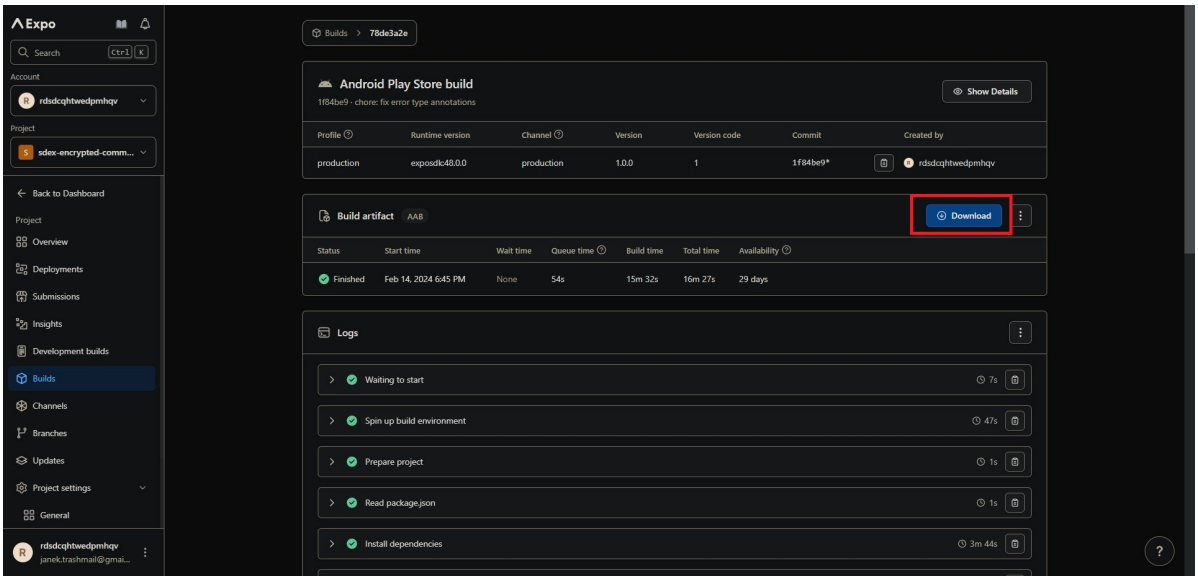
```
$ eas build -p android --profile production
```

następnie przejść na stronę: `https://expo.dev/accounts/<login>` (dostępną po zalogowaniu) i pobrać utworzony przez usługę Expo EAS plik instalacyjny `.apk` (Rysunek 4.1 i 4.2)



Rysunek 4.1: Widok strony z dostępnymi zbudowanymi wersjami aplikacji

Źródło: opracowanie własne



Rysunek 4.2: Widok strony ze szczegółami wybranej wersji (buildu) aplikacji i linkiem do pobrania

Źródło: opracowanie własne

Pobrany tym sposobem plik należy wgrać na swoje urządzenie i wybrać go w menadżerze plików na telefonie. Plik `.apk` jest wykonywalny i należy go zainstalować.

Istnieje również możliwość uruchomienia aplikacji w emulatorze. W tym celu należy mieć zainstalowane oprogramowanie Android Studio[6] oraz utworzone urządzenie wirtualne (ang. *Android Virtual Device*, AVD), a także biblioteki wymienione na początku tego paragrafu. Wówczas aplikację mobilną możemy uruchomić w powłoce (znajdując się w katalogu głównym aplikacji mobilnej) poprzez wywołanie komendy:

```
1    $ npx expo run:android ——device <nazwa emulatora>
```

Rozdział 5

Podsumowanie

Osiągnięte zostały cele pracy zdefiniowane w sekcji 1.3. Zostały stworzone aplikacje mobilna i serwerowa oferujące funkcjonalności wymienione w sekcjach 1.4.1 i 1.4.2, oraz wyciągnięto na podstawie tego wnioski opisane w sekcji 5.1.

5.1 Wnioski po stworzeniu aplikacji

- Wykorzystane algorytmy SDEx są stosunkowo proste w implementacji z wykorzystaniem funkcji bibliotecznej do generowania hashu oraz podstawowych funkcji oferowanych przez język programowania JavaScript (operacje na listach, operacja XOR).
- SDEx jest do pewnego stopnia skalowalny w zakresie długości bloku tekstu podlegającego pojedynczej iteracji szyfrowania / deszyfrowania (ograniczeniem jest tu maksymalna długość hashu jaką można uzyskać) oraz ograniczony jedynie sprzętowo pod względem całkowitej długości tekstu do zaszyfrowania / deszyfrowania - liczba iteracji jest nieograniczona od góry.
- Nowoczesne technologie mobilne pozwalają w prosty sposób wykorzystywać różnorakie funkcjonalności nowoczesnych telefonów (dostęp do pamięci, aparatu, zarządzanie uprawnieniami, udostępnianie plików przy pomocy zainstalowanych aplikacji) dzięki licznym bibliotekom i platformom takim jak Expo.
- Środowisko mobilne stawia pewne wyzwania, szczególnie dla aplikacji wymagającej podtrzymywania połączenia z serwerem do realizacji jej kluczowych funkcjonalności. Możliwość zerwania połączenia oraz utracenia połączenia z lokalną bazą danych wymagają sprawdzania dostępów do tych zasobów w wielu miejscach w aplikacji, co nie stanowi w takim stopniu ryzyka w aplikacji działającej na serwerze.
- Architektura aplikacji działającej w oparciu o interakcję z użytkownikiem w czasie rzeczywistym znacząco różni się od architektury aplikacji serwerowej, która z założenia powinna funkcjonować bez obsługi człowieka.

5.1.1 Ocena możliwości wdrożenia zaimplementowanego rozwiązania do środowiska produkcyjnego

W obecnej formie aplikacja nie nadaje się do wdrożenia na środowisku produkcyjnym, mimo że spełnia postawione wymogi funkcjonalne i oferuje bezpieczną wymianę danych

między serwerem i klientem mobilnym. Serwer daje taką możliwość, ale nie został skonfigurowany pod kątem obsługi jednoczesnego ruchu z bardzo wielu urządzeń.

Baza danych SQLite nie daje możliwości zarządzania dostępami dla różnych użytkowników, co w środowisku produkcyjnym również mogłoby być nieakceptowalne.

Klucze RSA serwera przechowywane są lokalnie w niezaszyfrowanych plikach tekstowych, a ich zawartość wczytywana do użytku przez serwer. W rozwiązaniu produkcyjnym klucze powinny być przechowywane w formie zaszyfrowanej lub pobierane z usługi zewnętrznej.

Bibliografia

[1] Ably. *WebSocket vs REST use cases*. URL: https://ably.com/topic/websocket-vs-rest. (dostęp: 05.02.2024).

[2] Rajesh Bondugula. *RSA public key: Behind the scene*. URL: https://medium.com/@bn121rajesh/understanding-rsa-public-key-70d900b1033c. (dostęp: 05.02.2024).

[3] DENSO WAVE INCORPORATED. *QRCode.com*. URL: https://www.qrcode.com/en/about/error_correction.html. (dostęp: 05.02.2024).

[4] Enlyft. *Companies using React Native*. URL: https://enlyft.com/tech/products/react-native. (dostęp: 05.02.2024).

[5] Sébastien Eustace. *Poetry - Python packaging and dependency management made easy*. URL: https://python-poetry.org/. (dostęp: 05.02.2024).

[6] Google. *Android Studio*. URL: https://developer.android.com/studio. (dostęp: 05.02.2024).

[7] Artur Hłobaż. „Analysis of the Possibility of Using Selected Hash Functions Submitted for the SHA-3 Competition in the SDEx Encryption Method". W: *INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS* 68 (sty. 2022), s. 60–61. DOI: 10.24425/ijet.2022.139848.

[8] Ruben Horn i in. „Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts". W: (2023), s. 5–7.

[9] KMS SOLUTIONS. *Mobile Application Development: A Comprehensive Guide*. URL: https://blog.kms-solutions.asia/an-ultimate-guide-to-mobile-application-development. (dostęp: 06.02.2024).

[10] Piotr Milczarski Krzysztof Podlaski Artur Hłobaż. „Secure Data Exchange Based on Social Network Public Key Distribution". W: (2016), s. 56–57.

[11] Meta Open Source. *Meta Open Source - About*. URL: https://opensource.fb.com/about. (dostęp: 05.02.2024).

[12] Meta Platforms, Inc. *React Native*. URL: https://reactnative.dev/. (dostęp: 08.02.2024).

[13] Tomas Vidhall Niclas Hansson. „Effects on performance and usability for cross-platform application development using React Native". Prac. mag. Linköping University, Department of Computer i Information Science, Human-Centered systems, 2016, s. 34–43.

[14] Krzysztof Podlaski Piotr Milczarski Artur Hłobaż. „Analysis of enhanced SDEx method". W: *The 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)* 68 (wrz. 2017), s. 1047–1048. DOI: 10.1109/IDAACS.2017.8095245.

[15] Jeff Carnahan (Windows Apps Team). *When to use a HTTP call instead of a WebSocket (or HTTP 2.0)*. URL: https://blogs.windows.com/windowsdeveloper/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/. (dostęp: 05.02.2024).

[16] TechEmpower. *Web Framework Benchmarks*. URL: https://www.techempower.com/benchmarks/#section=data-r22&hw=ph&test=fortune&l=zijzen-sf. (dostęp: 05.02.2024).

[17] Webiotic. *Different Types of Mobile App Development*. URL: https://www.webiotic.com/different-types-of-mobile-app-development/. (dostęp: 06.02.2024).

Spis rysunków

Spis listingów kodu