



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Technologie sieci urządzeń mobilnych oraz chmury obliczeniowej

Paweł Barczykowski

s22943

Aureliusz Sobieraj

s22925

Bezpieczny zdecentralizowany komunikator oparty na sieci Thread i najnowszych standardach kryptograficznych

Rodzaj pracy: praca magisterska

Promotor:

dr inż. Mariusz Trzaska, prof. PJATK

Warszawa, czerwiec 2026

*Pragniemy podziękować
promotorowi,
dr. inż. Mariuszowi Trzasce, prof. PJATK,
za nieocenioną pomoc
i wsparcie merytoryczne
podczas realizacji niniejszej pracy.*

*Dziękujemy naszym partnerkom
za wsparcie i cierpliwość
okazane podczas powstawania tej pracy.*

Streszczenie

Niniejsza praca przedstawia projekt oraz implementację zdecentralizowanego komunikatora tekstowego, zapewniającego szyfrowanie wiadomości od końca do końca odporne na ataki z wykorzystaniem komputerów kwantowych. System zbudowano w oparciu o sieć kratową działającą na protokole Thread (IEEE 802.15.4), co eliminuje konieczność stosowania centralnej infrastruktury serwerowej. Rolę węzłów sieci pełnią mikrokomputery Raspberry Pi z modułami radiowymi przeprogramowanymi na OpenThread, na których działa aplikacja backendowa napisana w języku Java z wykorzystaniem frameworka Spring Boot, reaktywnego stosu Spring WebFlux oraz interfejsu GraphQL. W odróżnieniu od istniejących rozwiązań komunikatorów zdecentralizowanych, system łączy sieć kratową o niskim poborze energii z kryptografią postkwantową, co stanowi podejście dotychczas niestosowane w tej klasie urządzeń.

Bezpieczeństwo komunikacji oparto na dwóch uzupełniających się mechanizmach kryptograficznych. Nawiązanie sesji realizowane jest przez protokół PQX3DH, stanowiący rozszerzenie standardowego protokołu X3DH o hybrydową wymianę kluczy z wykorzystaniem mechanizmu enkapsulacji Kyber-768, należącego do rodziny algorytmów postkwantowych zatwierdzonych przez NIST. Dalsza wymiana wiadomości zabezpieczona jest protokołem podwójnej zapadki, zapewniającym utajnianie z wyprzedzeniem oraz bezpieczeństwo po naruszeniu. Klucze kryptograficzne przechowywane są w Magazynie kluczy Android i chronione uwierzytelnieniem biometrycznym.

Klientem systemu jest natywna aplikacja na platformę Android, napisana w języku Kotlin, z interfejsem użytkownika zbudowanym w Jetpack Compose, komunikująca się z węzłem sieci za pomocą biblioteki Apollo Kotlin oraz przechowująca historię rozmów lokalnie w bazie Room.

Przeprowadzone testy w konfiguracji dwu- i trójwęzłowej potwierdziły stabilność i poprawność działania systemu. Ponad siedmiodniowy test długotrwałej pracy sieci zakończył się stuprocentowym wskaźnikiem dostarczenia wiadomości, a uzgodnienie klucza sesji PQX3DH wykazało pełną zgodność po obu stronach komunikacji we wszystkich przeprowadzonych próbach. Uzyskane wyniki potwierdzają, że połączenie zdecentralizowanej sieci Thread z postkwantowym szyfrowaniem od końca do końca stanowi realną alternatywę dla komunikatorów opartych na scentralizowanej infrastrukturze.

Słowa kluczowe: komunikacja zdecentralizowana, sieć kratowa, Thread, szyfrowanie od końca do końca, kryptografia postkwantowa, PQX3DH, Kyber-768, podwójna zapadka, Android, Spring Boot

Abstract

This thesis presents the design and implementation of a decentralized messaging system providing end-to-end encryption resistant to attacks carried out using quantum computers. The system is built on a mesh network operating according to the Thread protocol (IEEE 802.15.4), removing the need for centralized server infrastructure. Network nodes are implemented on Raspberry Pi single-board computers equipped with radio modules reflashed to run OpenThread, hosting a backend application written in Java using the Spring Boot framework, the reactive Spring WebFlux stack, and a GraphQL interface. Unlike existing decentralized messaging solutions, the system combines a low-power mesh network with post-quantum cryptography, an approach not previously applied in this class of devices.

Communication security relies on two complementary cryptographic mechanisms. Session establishment is performed using the PQX3DH protocol, an extension of the standard X3DH protocol incorporating a hybrid key exchange based on the Kyber-768 key encapsulation mechanism, one of the post-quantum algorithms standardized by NIST. Subsequent message exchange is protected by the Double Ratchet algorithm, providing forward secrecy and post-compromise security. Cryptographic keys are stored in the Android Keystore and protected by biometric authentication.

The client side of the system is a native Android application written in Kotlin, with a user interface built using Jetpack Compose, communicating with a network node through the Apollo Kotlin GraphQL client and storing conversation history locally in a Room database.

Tests carried out on two- and three-node configurations confirmed the stability and correctness of the system. A soak test exceeding seven days of continuous operation achieved a one-hundred-percent message delivery rate, while the PQX3DH session key agreement produced identical keys on both sides of the communication in every trial conducted. The obtained results demonstrate that combining a decentralized Thread mesh network with post-quantum end-to-end encryption constitutes a viable alternative to messengers based on centralized infrastructure.

Keywords: decentralized communication, mesh network, Thread, end-to-end encryption, post-quantum cryptography, PQX3DH, Kyber-768, Double Ratchet, Android, Spring Boot

Spis treści

1	Wstęp	1
1.1	Wprowadzenie do problematyki komunikacji zdecentralizowanej	1
1.2	Motywacja podjęcia tematu	1
1.3	Cel i zakres pracy	2
1.4	Rezultat pracy	2
1.5	Organizacja pracy	2
1.6	Podział pracy	2
2	Stan sztuki i analiza istniejących rozwiązań	4
2.1	Modele architektury systemów komunikacyjnych	4
2.2	Centralizacja vs. decentralizacja systemów	5
2.3	Technologie komunikacji w sieciach IoT	5
2.4	Przegląd współczesnych komunikatorów (Signal, WhatsApp, Telegram, Matrix)	6
2.5	Wady i ograniczenia istniejących rozwiązań	7
2.6	Założenia projektowe proponowanego systemu	7
3	Podstawy teoretyczne	9
3.1	Sieci rozproszone i architektura kratowa	9
3.2	Standard Thread (IEEE 802.15.4)	9
3.3	Podstawy kryptografii asymetrycznej	10
3.4	Bezpieczeństwo szyfrowania od końca do końca	10
3.5	Utajnianie z wyprzedzeniem i zarządzanie kluczami	11
4	Zastosowane technologie i narzędzia	12
4.1	Java	12
4.2	Spring Boot	12
4.3	Spring WebFlux	12
4.4	GraphQL	12
4.5	Altair GraphQL Client	13
4.6	Kotlin	14
4.7	Jetpack Compose	14
4.8	Apollo Kotlin	14
4.9	Room	14
4.10	Bouncy Castle	15
4.11	Magazyn kluczy Android	15
4.12	Gradle	15
4.13	Maven	15
4.14	IntelliJ IDEA	16
4.15	Android Studio	16
4.16	OpenThread	17
4.17	Gitea	17
5	Projekt systemu	19
5.1	Ogólna koncepcja systemu	19
5.1.1	Węzły sieci	20
5.1.2	Routing wiadomości	20
5.1.3	Klienci mobilni i model bezpieczeństwa	20
5.2	Architektura zdecentralizowana	21
5.2.1	Role węzłów w sieci Thread	21
5.2.2	Topologia testowa	21

5.2.3	Odporność na awarie	21
5.3	Warstwa komunikacyjna (OTCTL, Java)	21
5.3.1	Sterowanie siecią Thread — OtCtl	22
5.3.2	Transport UDP — NetService	22
5.3.3	Most między Thread a Spring — ThreadFluxBridge	22
5.4	Warstwa serwera (Spring Boot, GraphQL)	22
5.4.1	Schemat GraphQL	23
5.4.2	Resolwery GraphQL	23
5.4.3	Zarządzanie aktywnością użytkowników	24
5.4.4	Buforowanie wiadomości	24
5.5	Aplikacja mobilna (Android, Kotlin)	24
5.5.1	Architektura i stos technologiczny	24
5.5.2	Komunikacja z węzłem — Apollo GraphQL	24
5.5.3	Lokalne przechowywanie danych — Room	25
5.5.4	Moduł kryptograficzny — PQX3DH i podwójna zapadka	25
5.6	Model danych i przechowywanie informacji (Room)	26
5.6.1	Encje danych	26
5.6.2	Izolacja rozmów	26
5.6.3	Warstwa dostępu do danych	27
5.7	Mechanizmy bezpieczeństwa i szyfrowania (PQX3DH)	27
5.7.1	Uzgodnienie klucza sesji — PQX3DH	27
5.7.2	Szyfrowanie wiadomości — podwójna zapadka	28
5.7.3	Ochrona kluczy — Magazyn kluczy Android i biometria	28
6	Implementacja systemu	30
6.1	Środowisko implementacyjne	30
6.1.1	Środowisko sprzętowe	30
6.1.2	Backend	30
6.1.3	Aplikacja mobilna	30
6.2	Konfiguracja OpenThread i OTBR	31
6.2.1	Budowanie narzędzi OpenThread	31
6.2.2	Uruchamianie demona Thread	32
6.2.3	Zarządzanie siecią Thread z poziomu aplikacji	32
6.2.4	Konfiguracja punktu dostępowego Wi-Fi	33
6.2.5	Uruchamianie aplikacji	34
6.2.6	Panel administracyjny	34
6.3	Implementacja backendu	36
6.3.1	Spring WebFlux	37
6.3.2	Thread	40
6.3.3	ThreadFluxBridge	42
6.4	Implementacja aplikacji mobilnej	45
6.4.1	Room	45
6.4.2	Protokół PQX3DH	47
6.4.3	UI/UX	49
6.5	Implementacja GraphQL	56
6.5.1	Konfiguracja Apollo Kotlin	56
6.5.2	Operacje GraphQL po stronie klienta	57
6.5.3	Subskrypcja WebSocket w tle	58
6.5.4	Niezawodność połączenia	58
6.6	Narzędzia deweloperskie i pomocnicze	59
6.6.1	Programowe zarządzanie punktem dostępowym	59
6.6.2	Ochrona przed wielokrotnym uruchomieniem	60
6.7	Problemy napotkane podczas implementacji	60

6.7.1	Budowanie narzędzi OpenThread	61
6.7.2	Adresowanie IPv6 w sieci Thread	61
6.7.3	Autonomia protokołu Thread	61
6.7.4	Implementacja kryptografii na Androidzie	61
6.7.5	Konfiguracja punktu dostępowego Wi-Fi	62
7	Testy i badania	63
7.1	Testy stabilności sieci Thread	63
7.1.1	Formowanie sieci i czas gotowości	63
7.1.2	Re-elekcja Leadera	63
7.1.3	Długotrwała praca sieci	63
7.2	Testy aplikacji mobilnej	64
7.2.1	Reconnect WebSocket po utracie połączenia Wi-Fi	64
7.2.2	Restart usługi ChatService	64
7.2.3	Odporność na brak serwera przy starcie aplikacji	65
7.3	Testy poprawności szyfrowania	65
7.3.1	Weryfikacja uzgodnienia PQX3DH	65
7.3.2	Ciągłość sesji podwójnej zapadki	65
7.3.3	Weryfikacja ekranu odblokowania Magazynu kluczy	66
7.4	Testy wydajnościowe	66
7.4.1	Opóźnienie szyfrowania od końca do końca	66
7.4.2	Czas operacji kryptograficznych	66
7.5	Analiza wyników	67
7.5.1	Stabilność i niezawodność	67
7.5.2	Poprawność kryptograficzna	67
7.5.3	Wydajność	68
7.5.4	Kierunki dalszego rozwoju	68
8	Zastosowania i możliwości rozwoju	69
8.1	Potencjalne zastosowania systemu	69
8.1.1	Komunikacja w środowiskach korporacyjnych	69
8.1.2	Środowiska przemysłowe i Internet Rzeczy	69
8.1.3	Komunikacja kryzysowa	69
8.1.4	Zastosowania wojskowe i służby specjalne	69
8.2	Uniwersalność i skalowalność rozwiązania	70
8.3	Możliwości rozbudowy systemu	70
8.3.1	Przesyłanie zaszyfrowanych plików i multimediów	70
8.3.2	Zaawansowane mechanizmy samonaprawy połączenia	70
8.3.3	Skrócenie czasu rozruchu sieci	71
8.3.4	Uwierzytelnienie sprzętowe FIDO2	71
8.3.5	Formalna weryfikacja kryptograficzna	71
8.3.6	Dedykowane urządzenie końcowe oparte na mikrokontrolerze	71
9	Podsumowanie i wnioski	72
9.1	Podsumowanie osiągniętych rezultatów	72
9.2	Ocena realizacji celu pracy	72
9.3	Wnioski końcowe	73
	Bibliografia	74
	Spis Tabel	76
	Spis Rysunków	77
	Spis Listingów	78

Rozdział 1

Wstęp

Niniejsza praca dotyczy zaprojektowania i implementacji zdecentralizowanego systemu komunikacji opartego na sieci rozproszonej z wykorzystaniem protokołu Thread oraz kryptograficznych mechanizmów zapewniających bezpieczeństwo komunikacji końcowej. Poniższe podrozdziały wprowadzają w problematykę komunikacji zdecentralizowanej, przedstawiają motywację podjęcia tematu, określają cel i zakres pracy oraz opisują jej strukturę.

1.1 Wprowadzenie do problematyki komunikacji zdecentralizowanej

Większość współczesnych komunikatorów internetowych opiera się na modelu scentralizowanym, kontrolowanym przez duże korporacje technologiczne. Za pośrednictwem aplikacji takich jak WhatsApp [1], Telegram [2] czy Messenger [3] codziennie przesyłane są miliony wiadomości, jednak wszystkie dane przechodzą przez centralne serwery zarządzane przez dostawców usług. Choć takie rozwiązanie zapewnia wysoką wygodę użytkownika oraz upraszcza implementację systemu, wiąże się również z licznymi zagrożeniami dotyczącymi prywatności, bezpieczeństwa oraz niezależności użytkowników.

Centralizacja systemów komunikacyjnych prowadzi do powstania pojedynczego punktu awarii. Oznacza to, że uszkodzenie lub celowe wyłączenie infrastruktury operatora może skutkować całkowitą utratą możliwości komunikacji. Zjawisko to występuje zarówno w przypadku problemów technicznych, jak i działań o charakterze politycznym, gdy państwa ograniczają obywatelom dostęp do wybranych komunikatorów internetowych. Dodatkowo model scentralizowany wiąże się z ryzykiem masowej inwigilacji oraz gromadzenia danych użytkowników, nawet wtedy, gdy sama treść wiadomości pozostaje szyfrowana.

Odpowiedzią na opisane problemy jest komunikacja zdecentralizowana, w której ruch sieciowy nie jest obsługiwany przez pojedynczą instytucję, lecz rozproszony pomiędzy węzły tworzące sieć. Systemy tego typu charakteryzują się większą odpornością na awarie, ponieważ wyłączenie pojedynczego węzła nie powoduje zatrzymania działania całej infrastruktury. Dodatkowo brak centralnego punktu przechowywania danych istotnie ogranicza ryzyko związane z naruszeniem bezpieczeństwa oraz nieautoryzowanym dostępem do informacji. Przykładami praktycznych realizacji tej koncepcji są komunikatory Session [4] oraz Jami [5].

Szczególnym przykładem komunikacji zdecentralizowanej są sieci kratowe, wykorzystujące technologie radiowe krótkiego zasięgu. Rozwiązania tego typu umożliwiają wymianę informacji w środowiskach pozbawionych tradycyjnej infrastruktury internetowej, między innymi w obiektach przemysłowych. Ich potencjał jest szczególnie widoczny w kontekście Internetu Rzeczy, gdzie urządzenia dysponujące ograniczonymi zasobami obliczeniowymi i energetycznymi muszą komunikować się w sposób niezawodny oraz bezpieczny.

Decentralizacja nie rozwiązuje jednak problemu prywatności komunikacji. Sieć pozbawiona centralnego administratora wciąż może być narażona na podsłuchiwanie ruchu pomiędzy węzłami. Konieczne jest zatem zastosowanie mechanizmów kryptograficznych, zapewniających poufność wiadomości niezależnie od zaufania do poszczególnych węzłów sieci. Wzorcowym przykładem podejścia skupionego na szyfrowaniu jest komunikator Signal [6].

Zagadnienia te stanowią nieodłączny element projektowania bezpiecznych systemów zdecentralizowanych i zostaną szczegółowo omówione w dalszych rozdziałach niniejszej pracy.

1.2 Motywacja podjęcia tematu

Wybór tematu podyktowany był osobistym zainteresowaniem zagadnieniami bezpieczeństwa komunikacji oraz sieciami o ograniczonych zasobach, a także dostrzeżoną luką w istniejących rozwiązaniach. Współczesne komunikatory szyfrujące od końca do końca zakładają dostęp do infrastruktury internetowej i opierają bezpieczeństwo na algorytmach podatnych na ataki kwantowe. Z kolei systemy komunikacji rozproszonej przeznaczone dla środowisk Internetu Rzeczy koncentrują się na efektywności transmisji, pomijając kwestie prywatności na poziomie aplikacyjnym. Niniejsza praca stanowi próbę wypełnienia tej

luki poprzez połączenie zdecentralizowanej sieci kratowej z szyfrowaniem od końca do końca odpornym na zagrożenia postkwantowe.

Dodatkową motywację stanowiła praktyczna użyteczność opracowywanego rozwiązania oraz złożoność techniczna przedsięwzięcia. Systemy komunikacji niezależne od centralnej infrastruktury mogą znaleźć zastosowanie w środowiskach przemysłowych czy służbach ratowniczych działających w terenie pozbawionym zasięgu. Zaprojektowanie systemu obejmującego heterogeniczne warstwy – sieć zdecentralizowaną, serwer backendowy i aplikację mobilną – przy zachowaniu spójnego modelu bezpieczeństwa stanowi wyzwanie inżynierskie, które dostarcza doświadczenia trudnego do uzyskania w ramach typowych zadań programistycznych.

1.3 Cel i zakres pracy

Celem pracy jest zaprojektowanie oraz implementacja zdecentralizowanego systemu komunikacji działającego w sieci rozproszonej opartej o protokół Thread, zapewniającego szyfrowanie wiadomości typu od końca do końca.

Projektowany system składa się z trzech współpracujących warstw. Pierwszą z nich jest warstwa komunikacyjna oparta na środowisku OpenThread oraz bibliotece OTCTL. Drugą stanowi serwer backendowy zrealizowany z wykorzystaniem technologii Spring Boot i interfejsu GraphQL. Trzeci element systemu stanowi natywna aplikacja mobilna dla systemu Android, napisana w języku Kotlin.

1.4 Rezultat pracy

Rezultatem pracy jest koncepcja zdecentralizowanego systemu komunikacji umożliwiającego bezpieczną wymianę wiadomości wraz z prototypem stanowiącym jej implementację referencyjną i weryfikację przyjętych założeń.

W ramach pracy zaimplementowano mechanizmy komunikacji pomiędzy urządzeniami działającymi w sieci rozproszonej, obsługę wymiany wiadomości w czasie rzeczywistym oraz szyfrowanie od końca do końca wykorzystujące nowoczesne algorytmy kryptograficzne. System został zaprojektowany z uwzględnieniem ograniczeń charakterystycznych dla środowisk Internetu Rzeczy, takich jak ograniczone zasoby obliczeniowe urządzeń oraz niestabilność infrastruktury sieciowej.

Efektem końcowym jest funkcjonalne rozwiązanie demonstracyjne potwierdzające możliwość budowy bezpiecznego i odpornego na awarie komunikatora działającego bez konieczności wykorzystywania scentralizowanej infrastruktury komunikacyjnej.

1.5 Organizacja pracy

Praca składa się z dziewięciu rozdziałów. Rozdział 2 przedstawia analizę istniejących rozwiązań w zakresie komunikacji zdecentralizowanej oraz technologii sieciowych stosowanych w Internecie Rzeczy. Rozdział 3 omawia podstawy teoretyczne dotyczące sieci rozproszonych, standardu Thread oraz mechanizmów kryptograficznych wykorzystywanych w komunikacji szyfrowanej od końca do końca. Rozdział 4 zawiera charakterystykę technologii i narzędzi zastosowanych przy realizacji systemu. Rozdział 5 przedstawia projekt zdecentralizowanego komunikatora, natomiast rozdział 6 opisuje jego implementację. Rozdział 7 prezentuje wyniki przeprowadzonych testów i badań. Rozdział 8 omawia potencjalne zastosowania systemu oraz możliwości jego dalszego rozwoju, a rozdział 9 zawiera podsumowanie pracy i wnioski końcowe.

1.6 Podział pracy

Praca została wykonana w równym stopniu przez obu autorów, zarówno w zakresie implementacji systemu, jak i redakcji niniejszego dokumentu. Poniżej przedstawiono orientacyjny podział odpowiedzialności w ramach projektu.

Aureliusz Sobieraj odpowiadał przede wszystkim za warstwę sieciową systemu – konfigurację protokołu Thread, integrację z OpenThread oraz niskopoziomą konfigurację węzłów Raspberry Pi, w tym konfigurację punktu dostępowego Wi-Fi. W zakresie implementacji zajął się warstwą komunikacji kra-

towej po stronie backendu – kodem Java odpowiedzialnym za obsługę protokołu Thread oraz modulem `ThreadFluxBridge` stanowiącym most między siecią kratową a warstwą aplikacyjną.

Paweł Barczykowski odpowiadał za wysokopoziomową warstwę backendową – implementację reaktywnego stosu Spring WebFlux, interfejsu GraphQL oraz subskrypcji WebSocket. Ponadto zajął się całością implementacji aplikacji mobilnej w języku Kotlin, w tym projektowaniem interfejsu użytkownika oraz implementacją mechanizmów kryptograficznych: protokołu PQX3DH, algorytmu Double Ratchet oraz integracji z Magazynem kluczy Android.

Elementy na styku obu warstw – schemat GraphQL stanowiący kontrakt między aplikacją mobilną a serwerem oraz format pakietów wymienianych między warstwą Thread a warstwą aplikacyjną – były projektowane wspólnie przez obu autorów.

Podział ten miał charakter orientacyjny. W trakcie realizacji projektu autorzy wzajemnie wspierali się w swoich obszarach odpowiedzialności, a ich wkład w powstanie systemu był równoważny. Dokumentacja techniczna sporządzana była wspólnie z wykorzystaniem samodzielnie hostowanego środowiska Overleaf [7].

Rozdział 2

Stan sztuki i analiza istniejących rozwiązań

Poniższe podrozdziały przedstawiają analizę współczesnych systemów komunikacyjnych ze szczególnym uwzględnieniem architektur zdecentralizowanych, technologii IoT oraz metod zapewniania bezpieczeństwa i prywatności użytkowników. Omówiono podstawowe modele architektury komunikacyjnej, różnice pomiędzy systemami scentralizowanymi i zdecentralizowanymi, a także przegląd wybranych komunikatorów internetowych. Na tej podstawie wskazano ograniczenia istniejących rozwiązań oraz sformułowano założenia projektowe proponowanego systemu.

2.1 Modele architektury systemów komunikacyjnych

Architektura systemu komunikacyjnego określa sposób wymiany danych pomiędzy urządzeniami oraz rolę poszczególnych elementów sieci. Współczesne systemy komunikacyjne opierają się na jednym z dwóch podstawowych modeli architektonicznych: scentralizowanym i zdecentralizowanym.

Najbardziej rozpowszechnionym podejściem jest architektura klient-serwer [8]. W takim modelu użytkownicy komunikują się za pośrednictwem centralnego serwera odpowiedzialnego za autoryzację, przechowywanie wiadomości, synchronizację danych oraz zarządzanie połączeniami. Rozwiązanie to jest powszechnie stosowane w popularnych komunikatorach internetowych, ponieważ upraszcza zarządzanie systemem oraz umożliwia łatwe skalowanie usług. Centralny serwer pełni jednak rolę pojedynczego punktu awarii. W przypadku problemów infrastrukturalnych lub ataku na serwer cała usługa może przestać działać.

Alternatywą dla modelu klient-serwer jest architektura równy z równym (P2P) [9]. W tym podejściu urządzenia komunikują się bezpośrednio między sobą, a rola centralnego serwera zostaje ograniczona lub całkowicie wyeliminowana. Każdy węzeł sieci może jednocześnie pełnić funkcję klienta i serwera. Rozwiązania P2P zwiększają odporność systemu na awarie oraz utrudniają centralną kontrolę nad przesyłanymi danymi. Z drugiej strony pojawiają się problemy związane z synchronizacją danych, odnajdywaniem użytkowników oraz utrzymaniem stabilnej komunikacji w dużych sieciach.

W systemach rozproszonych często wykorzystuje się architekturę hybrydową. Łączy ona cechy modelu scentralizowanego i zdecentralizowanego. Serwery odpowiadają przykładowo za proces uwierzytelniania użytkowników lub wymianę kluczy publicznych, natomiast sama komunikacja odbywa się bezpośrednio pomiędzy urządzeniami końcowymi. Takie rozwiązanie pozwala ograniczyć ruch przechodzący przez centralną infrastrukturę, zachowując jednocześnie wygodę użytkowania.

W ostatnich latach coraz większe znaczenie zyskują systemy przetwarzania brzegowego [10] oraz przetwarzania mglistego. Zakładają one przetwarzanie danych bliżej urządzeń końcowych zamiast w chmurze. W kontekście komunikatorów i systemów IoT oznacza to możliwość lokalnego przetwarzania wiadomości, przechowywania danych bezpośrednio na urządzeniach użytkowników oraz ograniczenia udziału zewnętrznych serwerów.

W projektach nastawionych na bezpieczeństwo szczególnie ważnym elementem architektury staje się sposób przechowywania danych. Coraz częściej stosuje się lokalne bazy danych działające bezpośrednio na urządzeniu użytkownika. Takie podejście zmniejsza ryzyko wycieku danych z centralnego serwera oraz zwiększa prywatność komunikacji. Jednocześnie wymaga ono odpowiedniego mechanizmu synchronizacji oraz bezpiecznej wymiany informacji pomiędzy urządzeniami.

Rozwój technologii IoT oraz urządzeń mobilnych sprawił, że współczesne systemy komunikacyjne muszą działać w środowisku heterogenicznym. Oznacza to konieczność obsługi różnych technologii transmisji danych, wielu typów urządzeń oraz dynamicznie zmieniającej się topologii sieci. W rezultacie coraz częściej projektuje się systemy zdolne do działania w sposób autonomiczny i odporny na chwilową utratę dostępu do Internetu.

2.2 Centralizacja vs. decentralizacja systemów

Centralizacja jest obecnie dominującym podejściem w projektowaniu systemów komunikacyjnych. W modelu tym cała komunikacja przechodzi przez centralną infrastrukturę zarządzaną przez operatora usługi. Pozwala to na łatwe administrowanie systemem, monitorowanie działania sieci oraz szybkie wdrażanie nowych funkcjonalności. Dodatkowo użytkownik końcowy nie musi zajmować się konfiguracją połączeń ani utrzymaniem infrastruktury.

Pomimo licznych zalet centralizacja wiąże się z wieloma problemami. Najważniejszym z nich jest zależność od jednego operatora. Awaria serwerów, problemy z dostępnością usług chmurowych lub cyberatak mogą spowodować całkowite unieruchomienie systemu. Dodatkowo centralny operator posiada dostęp do dużej ilości metadanych związanych z użytkownikami. Nawet jeśli treść wiadomości jest szyfrowana, nadal możliwe jest analizowanie informacji takich jak częstotliwość kontaktów, lokalizacja urządzeń czy czas aktywności użytkowników.

Decentralizacja stanowi próbę rozwiązania tych problemów [11]. W systemach zdecentralizowanych odpowiedzialność za działanie sieci jest rozłożona pomiędzy wiele niezależnych węzłów. Brak pojedynczego punktu awarii zwiększa odporność systemu oraz utrudnia przejęcie kontroli nad całą infrastrukturą. W przypadku komunikatorów zdecentralizowanych użytkownicy mogą komunikować się bez konieczności ciągłego pośrednictwa centralnego serwera.

Jednym z głównych wyzwań decentralizacji jest zarządzanie tożsamością użytkowników oraz odnajdywanie urządzeń w sieci. W systemach scentralizowanych zadanie to realizuje serwer, natomiast w architekturach kratowych konieczne jest zastosowanie dodatkowych mechanizmów, takich jak rozproszone tablice routingu, publikacja kluczy publicznych lub dynamiczne wykrywanie sąsiednich węzłów.

Kolejnym problemem jest synchronizacja danych. W przypadku systemów scentralizowanych serwer przechowuje aktualny stan wiadomości i historii komunikacji. W rozwiązaniach zdecentralizowanych dane mogą znajdować się wyłącznie na urządzeniach użytkowników, co zwiększa prywatność, ale utrudnia odzyskiwanie danych po utracie urządzenia.

Istotnym aspektem jest również bezpieczeństwo. Systemy zdecentralizowane są zwykle bardziej odporne na masowe wycieki danych, ponieważ informacje nie są przechowywane w jednym miejscu. Jednocześnie zwiększa się znaczenie zabezpieczenia urządzeń końcowych. Atak na pojedynczy węzeł może prowadzić do przejęcia lokalnie przechowywanych danych lub kluczy kryptograficznych.

W praktyce wiele współczesnych rozwiązań wykorzystuje podejście pośrednie. Serwery pełnią ograniczoną funkcję pomocniczą, natomiast właściwa komunikacja odbywa się pomiędzy urządzeniami użytkowników. Taki model pozwala połączyć wygodę systemów centralnych z większą prywatnością oraz odpornością charakterystyczną dla systemów zdecentralizowanych.

2.3 Technologie komunikacji w sieciach IoT

Systemy Internetu Rzeczy wykorzystują wiele technologii komunikacyjnych różniących się zasięgiem, poborem energii, przepustowością oraz sposobem organizacji sieci. Dobór odpowiedniego rozwiązania zależy od wymagań konkretnego systemu.

Jedną z najpopularniejszych technologii komunikacyjnych jest Wi-Fi. Zapewnia ono wysoką przepustowość transmisji danych oraz łatwą integrację z istniejącą infrastrukturą sieciową. Wi-Fi jest powszechnie wykorzystywane w urządzeniach mobilnych oraz inteligentnych urządzeniach domowych. Wadą tego rozwiązania jest stosunkowo wysokie zużycie energii oraz ograniczona skalowalność w przypadku dużej liczby urządzeń.

W systemach IoT bardzo często stosowany jest również Bluetooth Low Energy (BLE). Technologia ta została zaprojektowana z myślą o energooszczędnej komunikacji na krótkim dystansie. BLE jest wykorzystywane między innymi w urządzeniach ubieralnych, sensorach oraz systemach automatyki domowej. Ograniczeniem jest jednak niewielki zasięg oraz niższa przepustowość w porównaniu do Wi-Fi.

Duże znaczenie w rozwoju nowoczesnych sieci IoT zyskał protokół Zigbee. Umożliwia on tworzenie sieci rozproszonej, w których urządzenia mogą przekazywać dane pomiędzy sobą. Dzięki temu zwiększa

się zasięg komunikacji oraz odporność na awarie pojedynczych węzłów. Zigbee charakteryzuje się niskim poborem energii i jest szeroko stosowany w systemach inteligentnego domu.

Rozwinięciem koncepcji sieci zdecentralizowanej jest protokół Thread [12, 13]. Został on zaprojektowany z myślą o bezpiecznej i energooszczędnej komunikacji urządzeń IoT. Thread bazuje na standardzie IEEE 802.15.4 [14] oraz wykorzystuje protokół IPv6, co ułatwia integrację z nowoczesnymi systemami sieciowymi. W przeciwieństwie do wielu starszych rozwiązań Thread umożliwia bezpośrednią adresację urządzeń w sieci IP.

Istotną cechą Thread jest brak konieczności stosowania centralnego koncentratora zarządzającego całą siecią. Węzły komunikują się między sobą w modelu rozproszonym, przekazując pakiety danych pomiędzy kolejnymi urządzeniami. Dzięki temu sieć może dynamicznie dostosowywać trasę transmisji do aktualnego stanu połączeń.

Thread został zaprojektowany z uwzględnieniem bezpieczeństwa komunikacji. Wykorzystuje mechanizmy szyfrowania AES oraz bezpieczną autoryzację urządzeń dołączających do sieci [12]. Technologia ta znajduje obecnie zastosowanie między innymi w ekosystemie Matter rozwijanym przez największe firmy technologiczne.

W systemach wykorzystujących wiele technologii komunikacyjnych istotną rolę odgrywają urządzenia typu most lub bramka. Pozwalają one na komunikację pomiędzy różnymi standardami transmisji, przykładowo pomiędzy Wi-Fi a Thread. Dzięki temu możliwe jest połączenie urządzeń mobilnych z siecią IoT bez konieczności wyposażania każdego urządzenia w obsługę wszystkich protokołów.

W kontekście projektowanego systemu komunikacyjnego szczególnie istotne jest połączenie urządzeń mobilnych z siecią kratową opartą o Thread. Takie rozwiązanie umożliwia ograniczenie zależności od infrastruktury internetowej oraz pozwala na budowę zdecentralizowanej sieci komunikacyjnej działającej nawet przy ograniczonym dostępie do Internetu.

2.4 Przegląd współczesnych komunikatorów (Signal, WhatsApp, Telegram, Matrix)

Współczesne komunikatory internetowe koncentrują się głównie na wygodzie użytkowania, wysokiej dostępności usług oraz bezpieczeństwie transmisji danych. Najpopularniejsze rozwiązania wykorzystują szyfrowanie od końca do końca, jednak ich architektura nadal w dużej mierze opiera się na centralnej infrastrukturze serwerowej.

Jednym z najczęściej wskazywanych przykładów komunikatora nastawionego na prywatność jest Signal. Aplikacja wykorzystuje protokół Signal Protocol, który zapewnia szyfrowanie od końca do końca wiadomości tekstowych, połączeń głosowych oraz przesyłanych plików [6, 15, 16]. Signal przechowuje ograniczoną ilość danych o użytkownikach i jest rozwijany jako oprogramowanie otwarcie źródłowe. Pomimo wysokiego poziomu bezpieczeństwa system nadal korzysta z centralnych serwerów odpowiedzialnych za obsługę komunikacji i powiadomień.

Kolejnym popularnym rozwiązaniem jest WhatsApp. Komunikator również wykorzystuje Signal Protocol do szyfrowania wiadomości [1, 17]. Dzięki integracji z ekosystemem firmy Meta aplikacja osiągnęła bardzo dużą popularność na całym świecie. Jednocześnie pojawiają się kontrowersje związane z gromadzeniem metadanych oraz powiązaniem komunikatora z innymi usługami należącymi do tego samego operatora.

Telegram stosuje odmienne podejście do bezpieczeństwa komunikacji. Domyślnie wiadomości przechowywane są w chmurze operatora i synchronizowane pomiędzy urządzeniami użytkownika. Szyfrowanie od końca do końca dostępne jest wyłącznie w trybie sekretnej rozmowy [18]. Telegram oferuje wysoką wydajność oraz rozbudowane funkcje społecznościowe, jednak model przechowywania danych budzi wątpliwości wśród części ekspertów zajmujących się cyberbezpieczeństwem.

Interesującym przykładem systemu zdecentralizowanego jest Matrix. Matrix stanowi otwarty protokół komunikacyjny umożliwiający federację serwerów [19]. Użytkownicy mogą korzystać z różnych instancji serwerowych, które komunikują się ze sobą podobnie jak w przypadku poczty elektronicznej.

Matrix wspiera szyfrowanie od końca do końca oraz integrację z wieloma platformami komunikacyjnymi [19]. Pomimo zdecentralizowanego charakteru system nadal opiera się na serwerach pośredniczących.

Wspólną cechą większości współczesnych komunikatorów jest silne uzależnienie od infrastruktury internetowej oraz usług chmurowych. Nawet systemy określane jako zdecentralizowane zazwyczaj wymagają działania serwerów odpowiedzialnych za synchronizację danych, przekazywanie wiadomości lub przechowywanie historii komunikacji.

2.5 Wady i ograniczenia istniejących rozwiązań

Pomimo dużego rozwoju współczesnych komunikatorów nadal występuje wiele problemów związanych z prywatnością, bezpieczeństwem oraz zależnością od infrastruktury centralnej.

Największym ograniczeniem większości popularnych komunikatorów jest centralizacja usług. Nawet jeśli treść wiadomości jest szyfrowana, operator systemu nadal posiada dostęp do metadanych użytkowników. Informacje takie jak lista kontaktów, adresy IP, czas logowania czy częstotliwość komunikacji mogą stanowić cenne źródło danych analitycznych.

Kolejnym problemem jest przechowywanie danych w chmurze. Historia wiadomości często znajduje się na serwerach operatora lub w zewnętrznych usługach kopii zapasowej. W przypadku włamania do infrastruktury serwerowej może dojść do wycieku danych użytkowników lub metadanych komunikacyjnych.

Wiele komunikatorów wymaga również stałego dostępu do Internetu. Ogranicza to możliwość działania systemu w środowiskach o słabej infrastrukturze sieciowej lub podczas awarii usług internetowych. W praktyce użytkownik staje się zależny od infrastruktury operatorów telekomunikacyjnych oraz dostawców usług chmurowych.

Istotnym ograniczeniem jest także brak pełnej kontroli użytkownika nad własnymi danymi. Zamknięte rozwiązania komercyjne nie umożliwiają dokładnej weryfikacji sposobu przetwarzania informacji ani działania mechanizmów bezpieczeństwa. Problem ten jest szczególnie widoczny w aplikacjach rozwijanych przez duże korporacje technologiczne.

W systemach zdecentralizowanych pojawiają się natomiast inne trudności. Przykładowo rozwiązania federacyjne, takie jak Matrix, nadal wymagają utrzymania serwerów pośredniczących. W rezultacie użytkownik nadal musi zaufać administratorowi wybranej instancji.

Ograniczeniem wielu współczesnych komunikatorów jest również brak integracji z technologiami IoT i sieciami rozproszonymi. Większość aplikacji została zaprojektowana wyłącznie z myślą o klasycznej komunikacji internetowej. Powoduje to trudności w budowie lokalnych, autonomicznych systemów komunikacyjnych działających niezależnie od globalnej infrastruktury Internetu.

W kontekście bezpieczeństwa istotnym problemem pozostaje również przechowywanie kluczy kryptograficznych. W niektórych rozwiązaniach klucze mogą być synchronizowane z chmurą lub przechowywane w sposób zależny od operatora systemu. Zwiększa to ryzyko nieautoryzowanego dostępu do danych użytkownika.

2.6 Założenia projektowe proponowanego systemu

Opisywany system stanowi alternatywę dla klasycznych komunikatorów internetowych opartych na centralnej infrastrukturze serwerowej. Głównym celem projektu jest stworzenie zdecentralizowanego komunikatora zapewniającego wysoki poziom prywatności oraz minimalny udział zewnętrznych serwerów w procesie komunikacji.

System wykorzystuje aplikację mobilną działającą na systemie Android. Wszystkie wiadomości przechowywane są lokalnie na urządzeniu użytkownika z wykorzystaniem bazy danych Room [20]. Takie podejście ogranicza konieczność przechowywania danych w chmurze oraz zwiększa kontrolę użytkownika nad własnymi informacjami.

Istotnym elementem systemu są węzły pośredniczące pełniące funkcję mostu pomiędzy siecią Wi-Fi a protokołem Thread. Węzły te umożliwiają komunikację urządzeń mobilnych z siecią kratową opartą na Thread, co pozwala na tworzenie zdecentralizowanej infrastruktury komunikacyjnej działającej niezależnie od klasycznych serwerów internetowych.

Komunikacja pomiędzy węzłami odbywa się w sposób rozproszony. Każdy węzeł przekazuje pakiety danych do kolejnych urządzeń w sieci zdecentralizowanej, co zwiększa zasięg komunikacji oraz odporność systemu na awarie pojedynczych elementów infrastruktury.

Do identyfikacji użytkowników oraz zabezpieczenia komunikacji zastosowano kryptografię asymetryczną. Klient aplikacji publikuje w sieci swój klucz publiczny, który wykorzystywany jest przez innych użytkowników do inicjalizacji połączenia. Wymiana informacji niezbędnych do ustanowienia szyfrowanego kanału komunikacyjnego realizowana jest zgodnie z protokołem X3DH [15].

Wiadomości przesyłane są w formie zaszyfrowanej od końca do końca, tak, aby treść komunikacji była dostępna wyłącznie dla stron uczestniczących w rozmowie. Węzły pośredniczące odpowiadają jedynie za przekazywanie zaszyfrowanych pakietów danych, bez możliwości odczytu ich zawartości.

Przyjętym założeniem projektowym jest ograniczenie roli centralnych serwerów wyłącznie do funkcji pomocniczych lub całkowita rezygnacja z nich. Dzięki temu system zachowuje odporność na awarie infrastruktury oraz utrudnia masowe pozyskiwanie danych użytkowników.

System uwzględnia również pracę w środowiskach o ograniczonym dostępie do Internetu. Wykorzystanie sieci rozproszonej opartej na Thread pozwala utrzymać lokalną komunikację nawet w przypadku niedostępności zewnętrznych usług sieciowych.

Proponowane rozwiązanie łączy cechy systemów IoT, architektury zdecentralizowanej oraz nowoczesnych mechanizmów kryptograficznych. Celem projektu nie jest zastąpienie współczesnych komunikatorów, lecz stworzenie alternatywnego rozwiązania skoncentrowanego na prywatności, autonomii użytkownika oraz ograniczeniu zależności od centralnej infrastruktury.

Rozdział 3

Podstawy teoretyczne

Niniejszy rozdział wprowadza zagadnienia teoretyczne, które posłużyły za fundament pod projektowany system komunikacji. Opisano tutaj architekturę sieci rozproszonych, a także specyfikację standardu Thread, który odpowiada za łączność między węzłami. Dalej poruszono kwestie bezpieczeństwa – od podstawowych mechanizmów kryptograficznych, przez ideę szyfrowania od końca do końca, aż po mechanizm utajniania z wyprzedzeniem i zasady zarządzania kluczami, czyli standard we współczesnych komunikatorach internetowych.

3.1 Sieci rozproszone i architektura kratowa

Sieci rozproszone to model organizacji systemów, w którym cała funkcjonalność opiera się na współpracy wielu samodzielnych urządzeń. Główną różnicą w porównaniu do klasycznej architektury klient–serwer jest brak jednego centralnego punktu odpowiedzialnego za ruch sieciowy. W tym modelu poszczególne węzły mogą jednocześnie wysyłać, odbierać i przekazywać dane dalej [21].

Taka konstrukcja przekłada się przede wszystkim na znacznie większą odporność na awarie. Jeśli jeden element sieci ulegnie uszkodzeniu, reszta systemu wciąż może normalnie pracować. Eliminuje to problem tzw. pojedynczego punktu awarii (ang. single point of failure), który jest największą słabością systemów scentralizowanych.

Jednym z najefektywniejszych sposobów na wdrożenie takiego modelu w praktyce jest architektura kratowa. Urządzenia tworzą w niej sieć wzajemnych powiązań, dzięki czemu wiadomości mogą „skakać” po kolejnych węzłach, aż dotrą do celu. Oznacza to, że nadawca i odbiorca nie muszą znajdować się w swoim bezpośrednim zasięgu.

W teorii i praktyce wyróżnia się dwa główne warianty tej topologii: pełną oraz częściową. W topologii pełnej każde urządzenie jest połączone bezpośrednio ze wszystkimi pozostałymi. Daje to ogromną niezawodność, ale staje się nieefektywne i trudne do utrzymania przy większej skali. Dlatego znacznie częściej wybiera się topologię częściową, gdzie węzły komunikują się po prostu ze swoimi najbliższymi sąsiadami.

Kluczowym atutem sieci rozproszonej jest też zdolność do samonaprawy i automatycznej zmiany tras. Gdy jakieś połączenie zostanie zerwane, protokół routingu potrafi samoczynnie wyznaczyć alternatywną ścieżkę. Sieć działa więc nieprzerwanie, nawet przy utracie części infrastruktury.

Ze względu na te cechy, architektura kratowa doskonale sprawdza się w świecie Internetu Rzeczy (IoT). Urządzenia IoT często mają mały zasięg radiowy i ograniczoną moc obliczeniową. Przekazywanie danych przez sąsiednie węzły pozwala drastycznie zwiększyć zasięg całej sieci bez inwestowania w dodatkowe routery czy punkty dostępowe.

W przypadku tego projektu, model rozproszony pozwolił na stworzenie zdecentralizowanego środowiska, w którym węzły nie tylko przesyłają pakiety, ale też bezpośrednio obsługują użytkowników. Dzięki temu system jest stabilniejszy i niezależny od zewnętrznych, centralnych serwerów.

3.2 Standard Thread (IEEE 802.15.4)

Thread to nowoczesny protokół sieciowy stworzony specjalnie z myślą o urządzeniach Internetu Rzeczy. Projektowano go tak, aby umożliwić budowę bezpiecznych, bezprzewodowych sieci rozproszonych o bardzo niskim poborze energii [12].

U podstaw Thread leży standard IEEE 802.15.4, który odpowiada za warstwę fizyczną i łączy danych. To on definiuje parametry transmisji radiowej, strukturę pakietów oraz zasady dostępu do pasma. Kluczową zaletą tego standardu – zwłaszcza w zestawieniu z tradycyjnym Wi-Fi – jest minimalne zużycie prądu, co jest kluczowe dla urządzeń zasilanych bateryjnie [14].

To, co wyróżnia Thread na tle starszych technologii IoT, to natywna obsługa protokołu IPv6. Każde urządzenie w sieci dostaje swój własny, unikalny adres IP. Pozwala to na bezpośrednią komunikację przy

użyciu standardowych mechanizmów internetowych, bez potrzeby stawiania skomplikowanych bramek tłumaczących protokoły.

Wewnątrz sieci Thread węzły mogą przyjmować różne role, z których najważniejsze to Router, End Device oraz Leader. Routery zajmują się przekazywaniem ruchu, z kolei Leader zarządza całą topologią i koordynuje działanie sieci. Wszystko dzieje się automatycznie – jeśli Leader nagle ulegnie awarii, pozostałe węzły same wybierają nowego zarządcę, dzięki czemu użytkownik nawet nie zauważy przerwy w działaniu.

Pod kątem bezpieczeństwa, Thread oferuje wbudowane szyfrowanie AES. Każde urządzenie próbujące dołączyć do sieci musi najpierw przejść proces uwierzytelnienia. Warto jednak pamiętać, że te zabezpieczenia działają na poziomie warstwy sieciowej – chronią samą transmisję między węzłami, ale nie zastępują szyfrowania na poziomie aplikacji.

Warto dodać, że Thread zyskał mocno na znaczeniu po integracji ze standardem Matter, za którym stoją największe firmy na rynku elektroniki użytkowej. To sprawiło, że technologia ta szybko staje się rynkowym standardem. W tworzonej systemie Thread scala całą zdecentralizowaną strukturę, łącząc zalety topologii kratowej i wygodę, jaką daje pełne wsparcie dla IPv6.

3.3 Podstawy kryptografii asymetrycznej

Trudno wyobrazić sobie współczesne systemy bezpieczeństwa bez kryptografii asymetrycznej. W przeciwieństwie do klasycznych algorytmów symetrycznych, tutaj każde urządzenie operuje na parze powiązanych ze sobą kluczy: publicznym i prywatnym [22].

Zasada działania jest prosta: klucz publiczny można bezpiecznie udostępnić każdemu, natomiast prywatny musi pozostać wyłącznie do wiadomości właściciela. To, co zostanie zaszyfrowane kluczem publicznym, da się odczytać tylko i wyłącznie za pomocą pasującego klucza prywatnego.

Taka asymetria pozwala na realizację kluczowych funkcji ochronnych – od podpisu cyfrowego, przez szyfrowanie wiadomości, aż po bezpieczną wymianę kluczy sesyjnych. Jest to szczególnie przydatne, gdy dwoje użytkowników chce bezpiecznie ustalić wspólne hasło (sekret), mimo że wcześniej nigdy się nie kontaktowali i przesyłają dane jawnym kanałem.

W praktycznych implementacjach kryptografia asymetryczna służy głównie do otwarcia bezpiecznego połączenia. Kiedy obie strony uzgodnią już wspólny sekret, system przełącza się na szyfrowanie symetryczne. Jest ono o wiele szybsze i mniej obciąża procesor.

Bezpieczeństwo tych algorytmów opiera się na problemach matematycznych, które są trudne do rozwiązania w krótkim czasie (np. faktoryzacja dużych liczb w RSA czy logarytm dyskretny w krzywych eliptycznych). Wizja powstania komputerów kwantowych wymusiła jednak rozwój kryptografii postkwantowej (PQC). Nowe algorytmy opierają się na strukturach kratowych, które mają oprzeć się atakom kwantowym. Dobrym przykładem jest algorytm Kyber, oficjalnie rekomendowany przez NIST do bezpiecznej wymiany kluczy.

W opisywanym projekcie klucze asymetryczne są fundamentem tożsamości użytkowników. Pozwalają na ich pewną identyfikację i są niezbędne do bezpiecznego inicjowania każdej nowej rozmowy.

3.4 Bezpieczeństwo szyfrowania od końca do końca

Szyfrowanie od końca do końca to obecnie złoty standard, jeśli mówimy o zachowaniu prywatności w komunikacji. Cała idea sprowadza się do tego, że wiadomość jest szyfrowana na urządzeniu nadawcy i odszyfrowywana dopiero na sprzęcie odbiorcy [23].

Największą zaletą szyfrowania od końca do końca jest to, że żaden pośrednik – niezależnie czy to router sieci Thread, czy serwer pośredniczący – nie ma technicznej możliwości podejrzenia treści wiadomości. Wszystkie urządzenia uczestniczące w wymianie widzą jedynie nieczytelny ciąg znaków. Dzięki temu, nawet jeśli atakujący przejmie kontrolę nad infrastrukturą sieciową, treść rozmów pozostaje bezpieczna. Jedyne, do czego atakujący może uzyskać dostęp, to metadane (np. czas wysłania wiadomości czy rozmiar pakietu).

Nowoczesne aplikacje wykorzystujące szyfrowanie od końca do końca wdrażają też mechanizmy weryfikacji tożsamości, co pozwala upewnić się, że nie padliśmy ofiarą ataku typu man-in-the-middle

(podszybie się pod rozmówcę). Istotne jest tutaj całkowite odseparowanie bezpieczeństwa transportu od bezpieczeństwa samej treści. Sieć może mieć własne zabezpieczenia, ale aplikacja i tak niezależnie szyfruje swój ruch.

W projektowanym komunikatorze zasada ta jest rygorystycznie przestrzegana. Pakiety opuszczają smartfon czy komputer już w zabezpieczonej formie, a węzły sieci rozproszonej pełnią wyłącznie rolę „listonosza”, który przenosi dane bez wiedzy o ich zawartości.

3.5 Utajnianie z wyprzedzeniem i zarządzanie kluczami

Samo zaszyfrowanie wiadomości to za mało – system musi być odporny na sytuację, w której ktoś po pewnym czasie przejmie klucze kryptograficzne użytkownika. Do ochrony przed takimi scenariuszami służy mechanizm utajniania z wyprzedzeniem.

Gwarantuje on, że ewentualne złamanie lub kradzież aktualnego klucza nie pozwoli napastnikowi na odszyfrowanie wiadomości zarejestrowanych w przeszłości [15]. Aby to osiągnąć, rezygnuje się ze stałych kluczy na rzecz tzw. kluczy efemerycznych (tymczasowych). Są one generowane tylko na potrzeby danej sesji, a po jej zakończeniu bezpowrotnie kasowane z pamięci podręcznej urządzenia. Współczesne systemy idą o krok dalej i regularnie odświeżają klucze nawet w trakcie jednej rozmowy. Dobrym przykładem jest protokół podwójnej zapadki (znany z komunikatora Signal). W tym rozwiązaniu wysłanie każdej kolejnej wiadomości powoduje automatyczne przekręcenie „grzechotki” kryptograficznej i wygenerowanie nowego klucza. Jeśli nawet jedna wiadomość zostanie przechwycona i rozszyfrowana, reszta historii rozmowy pozostaje bezpieczna.

Z protokołem podwójnej zapadki wiąże się też pojęcie wstecznej tajności. Oznacza ono, że system potrafi samoczynnie „uzdrowić” swoje bezpieczeństwo po ataku. Jeśli napastnik poznał jeden z kluczy sesyjnych, ale kolejne iteracje wymiany kluczy przebiegną poprawnie, to nowa korespondencja znów będzie dla niego całkowicie niedostępna.

Ostatnim elementem systemu jest bezpieczne przechowywanie samych kluczy na urządzeniu. Muszą być one odizolowane od reszty systemu operacyjnego, co na platformach mobilnych realizuje się sprzętowo – na przykład przy pomocy Magazynu kluczy Android.

W architekturze opisywanego systemu do bezpiecznego startu sesji i wymiany pierwszych kluczy posłużył protokół PQX3DH (odporny na komputery kwantowe), natomiast za późniejsze zachowanie utajnienia z wyprzedzeniem odpowiada podwójna zapadka. Klucze prywatne nigdy nie opuszczają urządzeń użytkowników, co minimalizuje ryzyko ich wycieku.

Wszystkie opisane w tym rozdziale mechanizmy i standardy teoretyczne tworzą spójną całość, która posłużyła do zbudowania bezpiecznego i stabilnego komunikatora. Ich praktyczną implementację oraz szczegóły techniczne opisano w kolejnych częściach pracy.

Rozdział 4

Zastosowane technologie i narzędzia

Niniejszy rozdział zawiera zwięzłe charakterystyki technologii i narzędzi wykorzystanych przy realizacji systemu, obejmujących zarówno platformę backendową opartą na ekosystemie Java, jak i warstwę mobilną zbudowaną w Kotlinie na platformę Android. Opisano również biblioteki kryptograficzne, narzędzia budowania oraz infrastrukturę sieciową i deweloperską.

4.1 Java

Java [24] to obiektowy język programowania stworzony przez Sun Microsystems w 1995 roku, obecnie rozwijany przez Oracle. Działa na wirtualnej maszynie Javy, co zapewnia przenośność kodu między różnymi platformami sprzętowymi i systemami operacyjnymi. Posiada automatyczne zarządzanie pamięcią przez mechanizm zbierania śmieci. Język jest silnie typowany i wspiera programowanie obiektowe, funkcyjne oraz współbieżne. Java 17 jest wersją posiadającą długoterminowe wsparcie.

W realizowanym projekcie Java stanowi podstawę warstwy serwerowej. Wybór wersji 17 LTS zapewnia stabilność środowiska uruchomieniowego przez wieloletni okres wsparcia, co jest istotne z punktu widzenia utrzymywalności systemu. Bogaty ekosystem bibliotek oraz dojrzałe narzędzia diagnostyczne ułatwiają wykrywanie problemów wydajnościowych i zarządzanie zasobami w środowisku produkcyjnym.

4.2 Spring Boot

Spring Boot to framework aplikacyjny dla języka Java, oparty na Spring Framework. Upraszcza tworzenie samodzielnych aplikacji serwerowych przez automatyczną konfigurację komponentów i wbudowany serwer HTTP. Artefaktem wynikowym jest wykonywalny plik JAR zawierający wszystkie zależności, co upraszcza wdrożenie aplikacji. Spring Boot dostarcza gotowe integracje z bazami danych, systemami kolejek, protokołami sieciowymi i interfejsami API [25].

Mechanizm autokonfiguracji Spring Boot znacząco skraca czas potrzebny na przygotowanie środowiska projektowego, eliminując konieczność ręcznego definiowania komponentów infrastrukturalnych. Aktuatory (*Actuators*) wbudowane w framework dostarczają gotowych punktów końcowych do monitorowania stanu aplikacji, co ułatwia obserwację systemu w środowisku wdrożeniowym. Dzięki profilom Spring możliwe jest utrzymanie oddzielnych konfiguracji dla środowisk deweloperskiego i produkcyjnego.

4.3 Spring WebFlux

Spring WebFlux to reaktywny stos webowy wchodzący w skład Spring Framework [26]. Oparty na bibliotece Project Reactor, implementuje model programowania reaktywnego zgodny ze specyfikacją Reactive Streams. Umożliwia obsługę wielu połączeń współbieżnych przy użyciu małej liczby wątków roboczych, co czyni go efektywnym rozwiązaniem dla aplikacji wymagających obsługi wielu jednoczesnych połączeń WebSocket.

W kontekście systemu obsługującego długotrwałe połączenia WebSocket podejście reaktywne ma bezpośrednie przełożenie na efektywność wykorzystania zasobów serwera. Model nieblokujący pozwala utrzymywać aktywne sesje bez rezerwowania osobnego wątku dla każdego klienta, co skutkuje przewidywalnym zużyciem pamięci nawet przy wzroście liczby równoczesnych połączeń. Typy *Mono* i *Flux* z biblioteki Project Reactor umożliwiają komponowanie złożonych przepływów danych w deklaracyjny i testowalny sposób.

4.4 GraphQL

GraphQL [27] to język zapytań do API oraz środowisko uruchomieniowe dla tych zapytań, opracowany przez Facebooka w 2012 roku i upubliczniony w 2015 roku. W przeciwieństwie do REST, klient

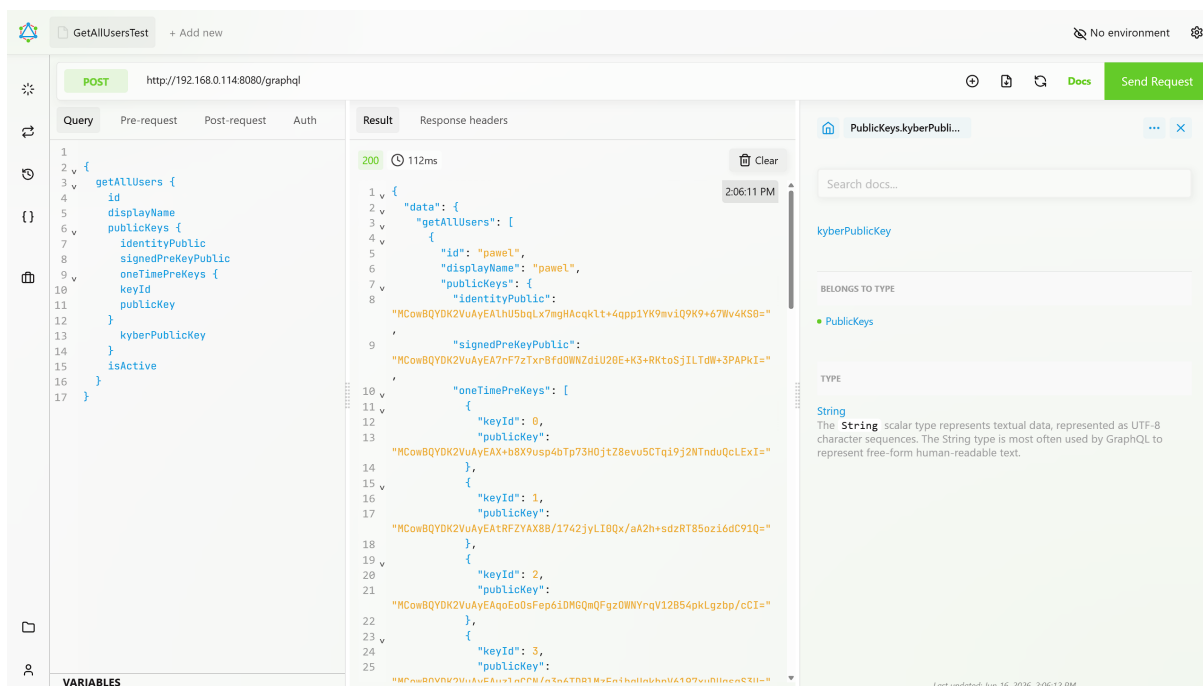
określa dokładnie jakich danych potrzebuje, co eliminuje problem nadmiarowego lub niewystarczającego pobierania danych. GraphQL definiuje trzy rodzaje operacji: zapytania (*Query*), mutacje (*Mutation*) oraz subskrypcje (*Subscription*), przy czym te ostatnie realizowane są przez protokół WebSocket i umożliwiają przesyłanie danych w czasie rzeczywistym.

Silnie typowany schemat GraphQL pełni rolę kontraktu między klientem a serwerem, umożliwiając wykrywanie niezgodności już na etapie kompilacji po stronie klienta. Introspekcja schematu pozwala narzędziom deweloperskim automatycznie generować dokumentację oraz podpowiedzi w środowiskach IDE. W projekcie subskrypcje GraphQL zostały wykorzystane do propagowania zdarzeń w czasie rzeczywistym do aplikacji mobilnej bez konieczności implementowania dedykowanego mechanizmu powiadomień po stronie serwera.

4.5 Altair GraphQL Client

Altair GraphQL Client [28] to graficzne narzędzie do interaktywnego testowania i eksploracji interfejsów API GraphQL, dostępne jako rozszerzenie przeglądarki oraz samodzielna aplikacja desktopowa. Umożliwia wysyłanie zapytań, mutacji i subskrypcji do dowolnego punktu końcowego GraphQL, prezentując wyniki w sformatowanej postaci JSON. Narzędzie obsługuje definiowanie nagłówków HTTP, zmiennych zapytań oraz środowisk (*environments*), co pozwala na wygodne przełączanie między konfiguracjami deweloperską i testową bez modyfikowania treści zapytań.

W trakcie realizacji projektu Altair służył jako podstawowe narzędzie weryfikacji poprawności schematu GraphQL oraz zachowania subskrypcji WebSocket. Wbudowana funkcja introspekcji schematu pozwala na automatyczne pobranie i przeglądanie dostępnych typów, zapytań i mutacji bezpośrednio z działającego serwera, co znacząco przyspiesza eksplorację API na etapie rozwoju. Historia wykonanych zapytań oraz możliwość ich zapisywania w kolekcjach ułatwia powtarzalne testowanie scenariuszy bez konieczności każdorazowego odtwarzania struktury żądania. Rysunek 4.1 przedstawia zrzut ekranu z wykonanego zapytania `getAllUsers`, pobierającego dane użytkowników wraz z ich kluczami kryptograficznymi: kluczem tożsamości (`identityPublic`), kluczem wstępnym (`signedPreKeyPublic`), kluczami jednorazowymi (`oneTimePreKeys`) oraz kluczem post-kwantowym Kyber (`kyberPublicKey`).



Rysunek 4.1: Zrzut ekranu z narzędzia Altair GraphQL Client z widocznym zapytaniem `getAllUsers`.

Źródło: Opracowanie własne

4.6 Kotlin

Kotlin [29] to statycznie typowany język programowania opracowany przez JetBrains, działający na JVM oraz kompilowany do kodu natywnego i JavaScript. Od 2017 roku jest preferowanym językiem do tworzenia aplikacji Android, rekomendowanym przez Google. Oferuje zwięzłą składnię, bezpieczeństwo typów nullable oraz korutyny jako wbudowany mechanizm programowania asynchronicznego. Korutyny pozwalają pisać kod asynchroniczny w stylu sekwencyjnym, bez konieczności stosowania klas zwrotnych.

Pełna interoperacyjność z Javą umożliwia stopniową migrację istniejących baz kodu oraz swobodne korzystanie z bibliotek dostępnych w ekosystemie JVM. Typy danych (*data classes*), klasy zapieczętowane (*sealed classes*) oraz destrukuryzacja pozwalają na zwięzłe modelowanie dziedziny aplikacji przy jednoczesnym zachowaniu pełnego bezpieczeństwa typów. Mechanizm rozszerzeń (*extension functions*) pozwala wzbogacać istniejące klasy o nową funkcjonalność bez modyfikowania ich kodu źródłowego.

4.7 Jetpack Compose

Jetpack Compose [30] to nowoczesny toolkit do budowania interfejsów użytkownika na platformę Android, opracowany przez Google. Zastępuje klasyczny system widoków oparty na XML deklaratywnym modelem opisywania interfejsu w języku Kotlin. Interfejs jest funkcją stanu – każda zmiana stanu powoduje automatyczne przeliczenie i ponowne wyrenderowanie tylko tych elementów, których stan uległ zmianie.

Ścisła integracja z korutinami Kotlina i biblioteką Flow upraszcza obsługę asynchronicznych źródeł danych bezpośrednio w warstwę prezentacji. System *Material Design 3* dostarczany wraz z biblioteką komponentów Compose zapewnia spójność wizualną zgodną z wytycznymi Google bez konieczności ręcznego implementowania stylistyki. Narzędzia podglądu (*Preview*) dostępne w Android Studio pozwalają weryfikować wygląd komponentów w czasie rzeczywistym bez uruchamiania aplikacji na urządzeniu lub emulatorze.

4.8 Apollo Kotlin

Apollo Kotlin to klient GraphQL dla języka Kotlin i platformy Android [31]. Generuje typowane klasy Kotlin na podstawie schematu GraphQL i zdefiniowanych operacji w czasie kompilacji, co eliminuje ręczne parsowanie odpowiedzi JSON. Obsługuje zapytania HTTP, mutacje oraz subskrypcje WebSocket. Wersja 4.x wprowadza w pełni korutynowe API oparte na Flow.

Normalizacja pamięci podręcznej (*normalized cache*) wbudowana w Apollo Kotlin pozwala na spójne zarządzanie danymi pobranymi z serwera po stronie klienta, eliminując duplikaty i umożliwiając aktualizację wielu widoków przy jednej operacji zapisu do cache'u. Mechanizm optymistycznych aktualizacji (*optimistic updates*) umożliwia natychmiastowe odzwierciedlanie zmian w interfejsie użytkownika przed potwierdzeniem operacji przez serwer, co poprawia responsywność aplikacji. Wtyczka Gradle generująca kod wykonywana jest w ramach standardowego procesu budowania, dzięki czemu typowane modele są zawsze zsynchronizowane z aktualną wersją schematu.

4.9 Room

Room to biblioteka trwałości danych dla platformy Android, wchodząca w skład Jetpack [20]. Stanowi warstwę abstrakcji nad bazą danych SQLite, dostarczając weryfikację zapytań SQL w czasie kompilacji i mapowanie wyników na obiekty Kotlin. Integruje się z korutinami i Flow, umożliwiając reaktywne obserwowanie zmian w bazie danych. Generowanie kodu DAO realizowane jest przez procesor adnotacji KSP (*Kotlin Symbol Processing*).

Mechanizm migracji bazy danych wbudowany w Room pozwala na bezpieczne wprowadzanie zmian schematu bez utraty danych użytkownika w kolejnych wersjach aplikacji. Obsługa relacji przez adnotacje `@Relation` i `@Embedded` umożliwia modelowanie złożonych struktur obiektowych przy jednoczesnym zachowaniu znormalizowanego schematu bazy danych. Testy jednostkowe zapytań DAO można przepro-

wadzać na wbudowanej bazie danych SQLite działającej w pamięci operacyjnej, co eliminuje zależność od fizycznego urządzenia w procesie weryfikacji logiki dostępu do danych.

4.10 Bouncy Castle

Bouncy Castle to otwarta biblioteka kryptograficzna dla języków Java i C# [32]. Dostarcza implementacje szerokiego zestawu algorytmów kryptograficznych, w tym wielu niedostępnych w standardowych bibliotekach platformy. Biblioteka jest certyfikowana według standardu FIPS 140-2 i regularnie aktualizowana o nowe algorytmy, w tym algorytmy post-quantowe zatwierdzone przez NIST. Wersja 1.82 wprowadza pełną implementację rodziny algorytmów CRYSTALS, w tym Kyber-768.

Algorytm Kyber-768, znany również jako ML-KEM, został wybrany przez NIST jako standard enkapsulacji klucza odporny na ataki z wykorzystaniem komputerów kwantowych. Jego zastosowanie w projekcie umożliwia ustanowienie bezpiecznego kanału wymiany kluczy symetrycznych nawet w obliczu potencjalnych zagrożeń kryptoanalizą kwantową. Połączenie Kyber-768 z klasycznymi algorytmami asymetrycznymi w schemacie hybrydowym zapewnia ochronę zarówno przed zagrożeniami klasycznymi, jak i post-quantowymi.

4.11 Magazyn kluczy Android

Magazyn kluczy Android [33] to podsystem bezpieczeństwa platformy Android umożliwiający generowanie i przechowywanie kluczy kryptograficznych w bezpiecznym środowisku sprzętowym, odizolowanym od reszty systemu operacyjnego. Klucze wygenerowane w Magazynie kluczy nigdy nie opuszczają tego środowiska w postaci jawnej. System obsługuje wymaganie uwierzytelnienia użytkownika (biometrycznego lub PIN) przed każdą operacją kryptograficzną, z konfigurowalnym oknem ważności uwierzytelnienia.

Na urządzeniach wyposażonych w dedykowany układ bezpieczeństwa (*Strongbox*) operacje kryptograficzne są izolowane sprzętowo nawet w przypadku kompromitacji głównego procesora aplikacyjnego. Atestacja kluczy (*key attestation*) umożliwia zdalne potwierdzenie serwerowi, że dany klucz został wygenerowany i jest przechowywany w certyfikowanym środowisku sprzętowym, co stanowi fundament mechanizmów weryfikacji integralności urządzenia. Cykl życia klucza, obejmujący generowanie, rotację i unieważnienie, jest zarządzany bezpośrednio przez system operacyjny, co minimalizuje ryzyko błędów implementacyjnych po stronie aplikacji.

4.12 Gradle

Gradle [34] to system automatyzacji budowania projektów, szeroko stosowany w ekosystemie Java i Android. Konfiguracja projektu opisywana jest w językach Groovy lub Kotlin DSL. Gradle obsługuje zarządzanie zależnościami, kompilację, uruchamianie testów i pakowanie artefaktów. W projektach Android jest domyślnym narzędziem budowania, integrującym się bezpośrednio z Android Studio i procesorem KSP.

Inkrementalne budowanie oraz pamięć podręczna zadań (*build cache*) znacząco skracają czas kolejnych kompilacji przez ponowne wykorzystanie wyników niezmiennych zadań. Konfiguracja katalogów wersji (*version catalogs*) wprowadzona w nowszych wersjach Gradle umożliwia centralne zarządzanie wersjami zależności w projektach wielomodułowych, eliminując rozbieżności między modułami. Wtyczki Gradle stosowane w projekcie, w tym wtyczka Apollo Kotlin do generowania kodu oraz KSP do przetwarzania adnotacji Room, są wykonywane automatycznie w ramach standardowego procesu budowania.

4.13 Maven

Maven [35] to narzędzie do automatyzacji budowania projektów Java, opracowane przez Apache Software Foundation. Konfiguracja projektu opisywana jest w pliku `pom.xml` w formacie XML. Maven zarządza zależnościami przez centralny repozytorium Maven Central, obsługuje cykl życia budowania (kompilacja, testowanie, pakowanie, wdrożenie) i jest szeroko stosowany w projektach enterprise.

4.14 IntelliJ IDEA

The screenshot displays an IDE interface with a Maven project. The left sidebar shows the project structure, including the 'thread-chat' module. The main editor area shows the 'pom.xml' file for the 'thread-chat' module. The right sidebar shows the Maven tool window with a tree view of the project structure.

Project Structure (Left Sidebar):

- OpenThreadCommunicator_maven
 - idea
 - mm
 - src
 - main
 - java
 - com.threadchat
 - bridge
 - ThreadFluxBridge
 - flux
 - controler
 - AdminControler
 - graphql
 - MessageResolver
 - UserResolver
 - model
 - Message
 - OneTimePreKeyInput
 - PublicKeys
 - User
 - service
 - MessageCache
 - MessageService
 - UserActivityManager
 - UserCleanupService
 - UserService
 - thread
 - BadMagicException
 - Config
 - CryptoUtils
 - KeyUtil
 - MagFrame
 - NetService
 - Octet
 - ParseFrame
 - Peer
 - ThreadChat
 - ThreadChatIntegrated
 - ThreadStateListener
 - ThreadStateMonitor

thread-chat/pom.xml (Main Editor):

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.threadchat</groupId>
  <artifactId>thread-chat</artifactId>
  <version>1.3.5</version>
  <packaging>jar</packaging>

  <properties>
    <java.version>17</java.version>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- Spring Boot -->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-graphql</artifactId>
    </dependency>

    <!-- Spring Boot WebFlux (Reactive) -->
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>

    <!-- Reactor Core -->
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    </dependency>

    <!-- Jackson dle JSON -->
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    </dependency>

    <!-- GraphQL Java Extended Scalars (apocymnie) -->
    <groupId>com.graphql-java-extended-scalars</groupId>
    <artifactId>graphql-java-extended-scalars</artifactId>
    <version>21.0</version>
    </dependency>
  </dependencies>

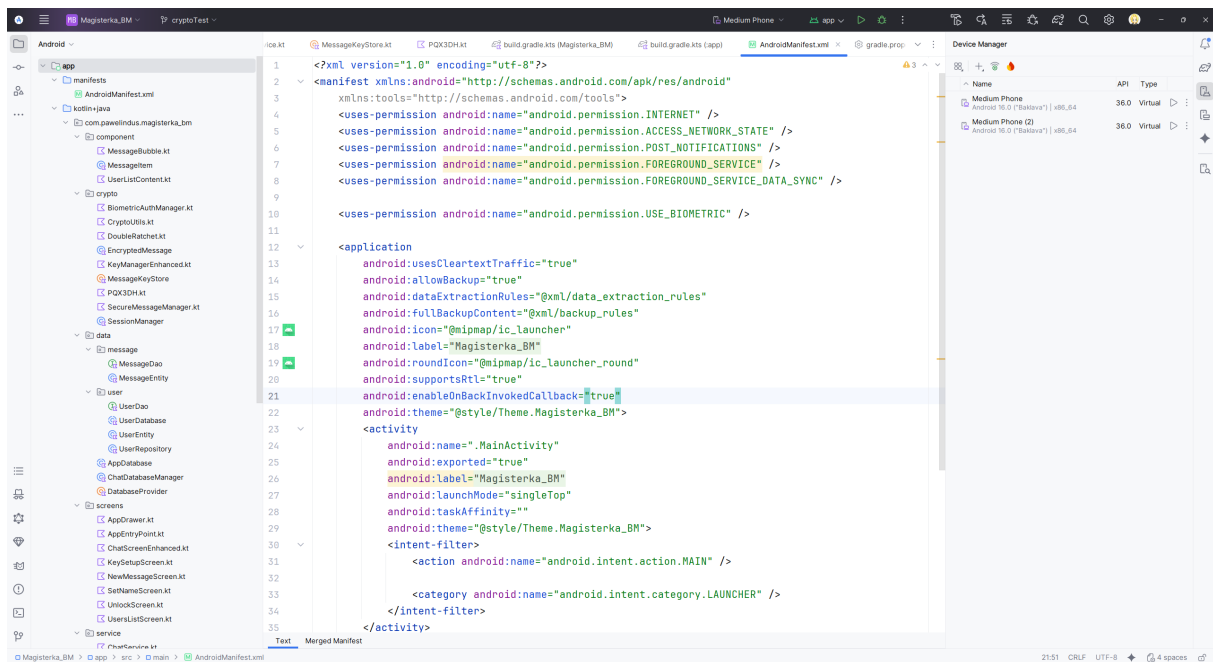
```

Maven Tool Window (Right Sidebar):

- Profiles
- thread-chat
 - Lifecycle
 - clean
 - validate
 - compile
 - test
 - package
 - verify
 - install
 - site
 - deploy
 - Plugins
 - Dependencies
 - Repositories

Wbudowany klient HTTP pozwala testować punkty końcowe REST i GraphQL bezpośrednio z poziomu IDE bez konieczności korzystania z zewnętrznych narzędzi. Głęboka integracja z frameworkiem Spring, obejmująca nawigację po ziarnach kontekstu aplikacji i inspekcję mapowań żądań HTTP, znacząco skraca czas diagnozowania problemów konfiguracyjnych. Narzędzia profilowania pamięci i wątków dostępne w wersji Ultimate umożliwiają wykrywanie wycieków zasobów oraz analizę zachowania aplikacji pod obciążeniem bez opuszczania środowiska deweloperskiego.

Android Studio [37] to oficjalne zintegrowane środowisko programistyczne do tworzenia aplikacji na platformę Android, opracowane przez Google na bazie IntelliJ IDEA. Dostarcza emulator urządzeń Android, narzędzia do profilowania wydajności i zużycia pamięci, wizualny edytor układów Compose oraz integrację z narzędziem budowania Gradle. Obsługuje debugowanie na fizycznych urządzeniach przez ADB (*Android Debug Bridge*). Rysunek 4.3 przedstawia rzut ekranu z programu z widoczną strukturą projektu i dostępnymi urządzeniami wirtualnymi.



Rysunek 4.3: Zrzut ekranu ze środowiska Android Studio. Źródło: Opracowanie własne

Narzędzie Layout Inspector umożliwia inspekcję hierarchii komponentów Compose w czasie działania aplikacji, co ułatwia diagnozowanie problemów z renderowaniem interfejsu użytkownika. Profiler aplikacji dostępny w Android Studio pozwala na szczegółową analizę zużycia procesora, pamięci, energii oraz ruchu sieciowego w ujęciu czasowym, umożliwiając identyfikację wąskich gardeł wydajnościowych. Emulator AVD (*Android Virtual Device*) obsługujący akcelerację sprzętową pozwala testować aplikację na wielu konfiguracjach systemowych bez dostępu do fizycznych urządzeń docelowych.

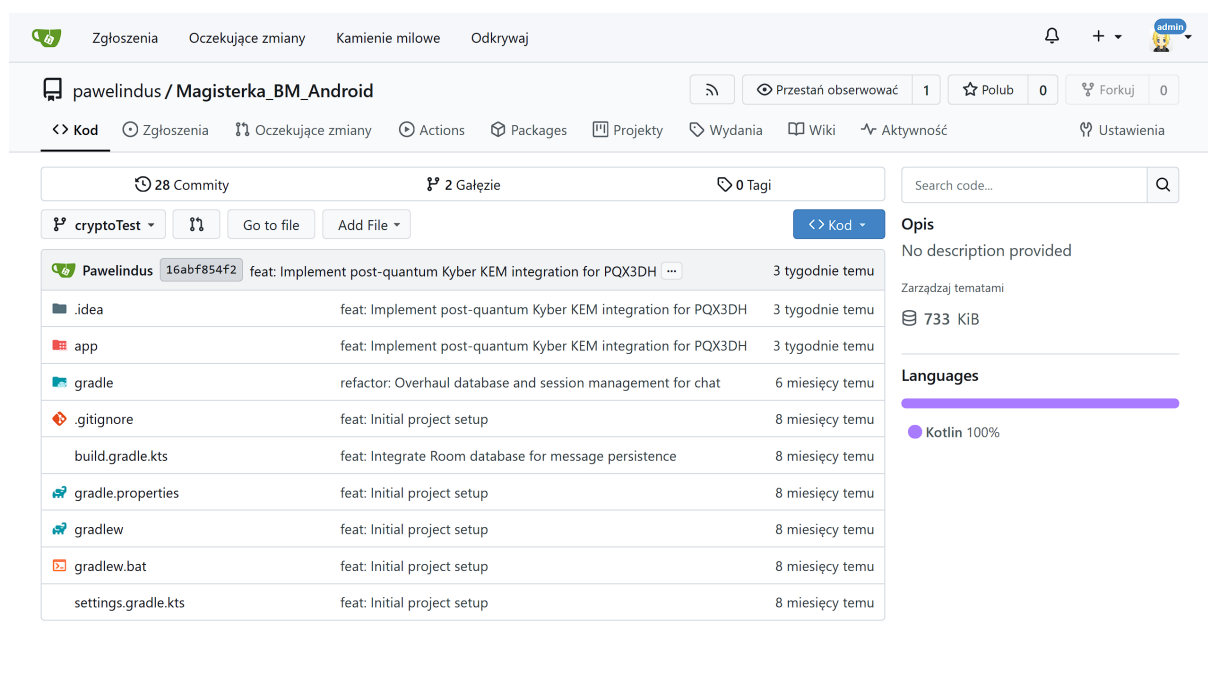
4.16 OpenThread

OpenThread to otwarta implementacja protokołu Thread, udostępniona przez Google na licencji BSD. Dostarcza stos sieciowy Thread działający na systemach operacyjnych klasy POSIX, w tym Linux. Projekt zawiera narzędzia `ot-daemon` (demon zarządzający interfejsem sieciowym Thread) oraz `ot-ctl` (narzędzie wiersza poleceń do sterowania demonem). OpenThread jest referencyjną implementacją protokołu Thread [13].

Protokół Thread buduje samonaprawiającą się sieć kratową (*mesh*) opartą na standardzie IEEE 802.15.4, zapewniając łączność nawet w przypadku awarii pojedynczych węzłów. Mechanizm wyboru lidera sieci (*leader election*) oparty na wartościach wagi węzłów (*router weight*) gwarantuje autonomiczną reorganizację topologii bez centralnego punktu zarządzania. Wsparcie dla IPv6 i protokołu CoAP wbudowane w stos OpenThread umożliwia bezpośrednią integrację urządzeń Thread z infrastrukturą siecią IP przez węzeł graniczny (*Border Router*).

4.17 Gitea

Gitea [38] to lekka, otwarta platforma do hostowania repozytoriów Git, napisana w języku Go. Oferuje interfejs webowy do przeglądania kodu, zarządzania repozytoriami, śledzenia zgłoszeń i przeglądania historii zmian. Gitea może być uruchomiona samodzielnie na własnej infrastrukturze (*self-hosted*), co zapewnia pełną kontrolę nad kodem źródłowym bez uzależnienia od zewnętrznych dostawców. Jest znacznie lżejsza zasobowo niż alternatywne rozwiązania takie jak GitLab. Rysunek 4.4 przedstawia rzut ekranu z widocznym repozytorium projektu aplikacji klienckiej na Androida.



Rysunek 4.4: Zrzut ekranu z repozytorium Gitea. Źródło: Opracowanie własne

Wbudowany system CI/CD (*Gitea Actions*), kompatybilny składniowo z GitHub Actions, umożliwia automatyzację procesu budowania i testowania bezpośrednio na własnej infrastrukturze bez przekazywania kodu do zewnętrznych usług. Obsługa webhooków pozwala na integrację Gitea z zewnętrznymi systemami powiadomień oraz narzędziami do zarządzania zadaniami, tworząc spójny przepływ pracy deweloperskiej. Niskie wymagania sprzętowe platformy pozwalają na jej uruchomienie na zasobooszczędnych serwerach, co jest szczególnie istotne w kontekście infrastruktury self-hosted.

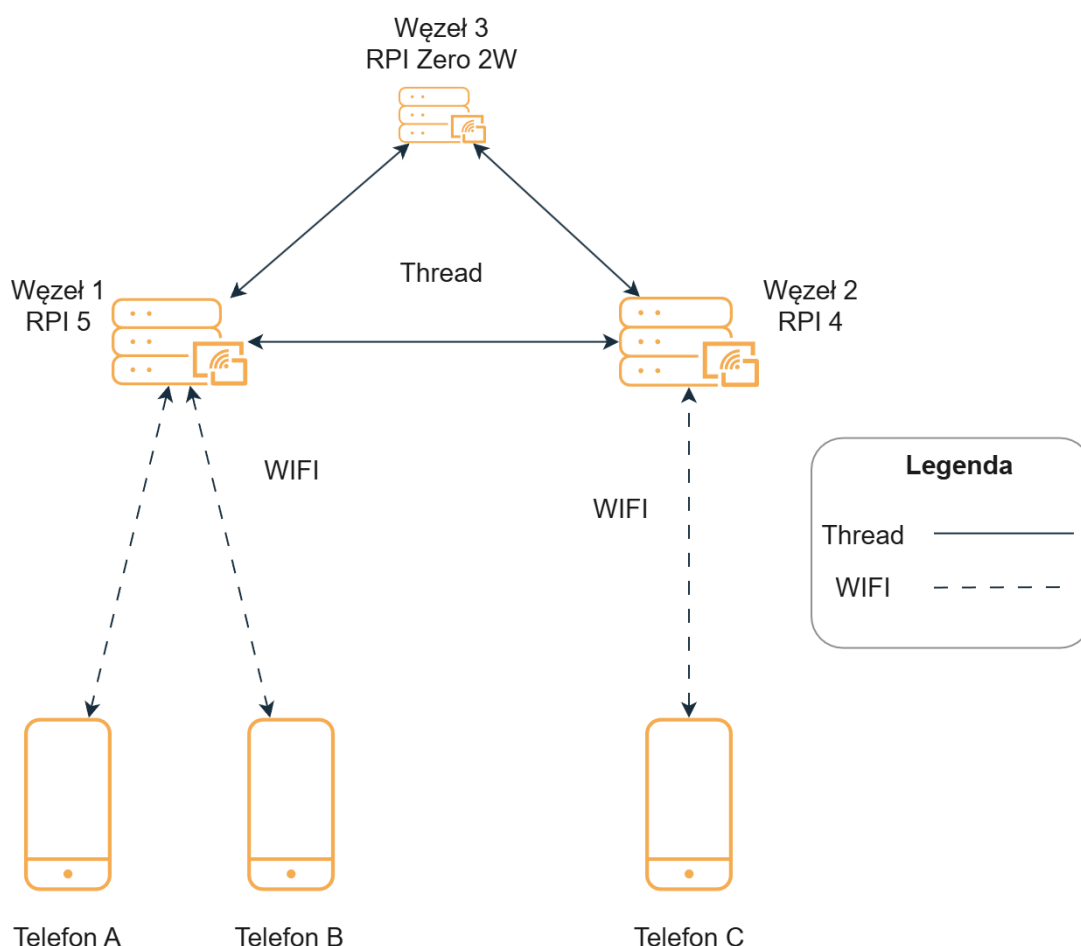
Rozdział 5

Projekt systemu

Niniejszy rozdział przedstawia projekt zdecentralizowanego komunikatora zbudowanego w oparciu o sieć rozproszoną z wykorzystaniem protokołu Thread. Omówiono strukturę węzłów, poszczególne warstwy systemu oraz mechanizmy bezpieczeństwa. Podrozdział 5.1 opisuje ogólną koncepcję systemu, podrozdziały 5.2–5.4 omawiają architekturę zdecentralizowaną oraz warstwy komunikacyjną i serwerową, a podrozdziały 5.5–5.7 przedstawiają aplikację mobilną, model danych oraz mechanizmy kryptograficzne.

5.1 Ogólna koncepcja systemu

Opisywany system to zdecentralizowany komunikator. Nie ma w nim centralnego serwera, a rolę węzłów pełnią mikrokomputery z modułami radiowymi przeprogramowanymi na OpenThread. Każdy węzeł pełni zarówno rolę koordynatora sieci Thread, jak również jest odpowiedzialny za obsługę klientów, rozpoczynanie komunikacji oraz przekazywanie wiadomości od i do klientów Android. Ogólną architekturę systemu przedstawia rysunek 5.1.



Rysunek 5.1: Ogólna architektura systemu komunikacji zdecentralizowanej.
Źródło: Opracowanie własne

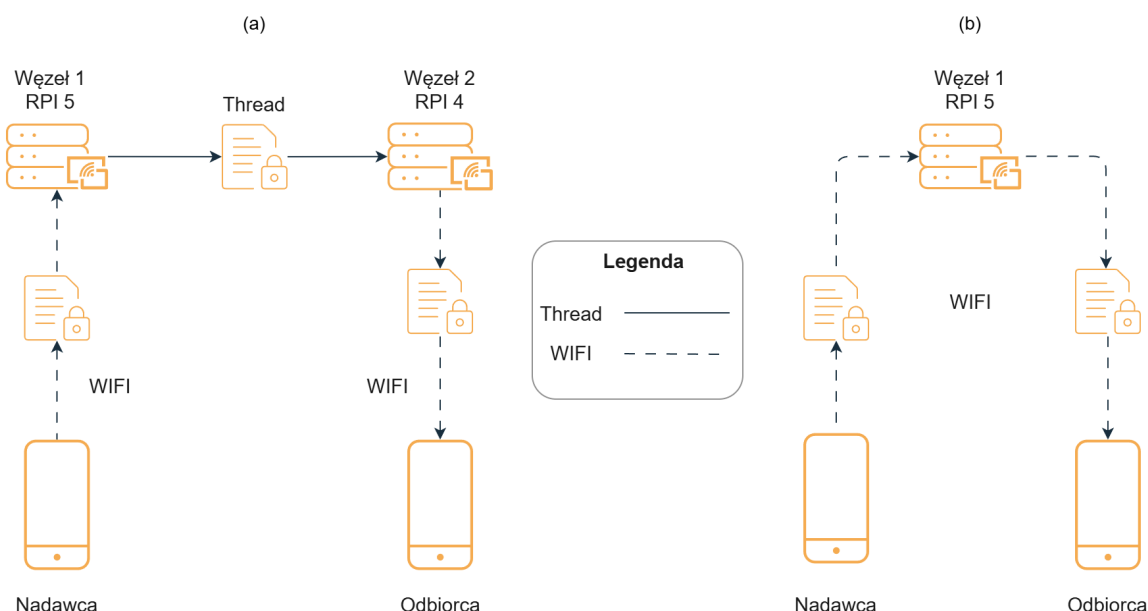
5.1.1 Węzły sieci

Na każdym węźle działają dwa komponenty. Pierwszym jest demon OpenThread Border Router (OTBR) z biblioteką OTCTL, który udostępnia stos Thread systemowi operacyjnemu. Drugim jest aplikacja backendowa w języku Java, oparta na Spring WebFlux i GraphQL. Wewnętrznie dzieli się ona na trzy moduły:

- *Thread* – sterujący demonem OTCTL przez wywołania poleceń;
- *ThreadFluxBridge* – tłumaczący komunikaty Thread na obiekty Javy eksponowane przez GraphQL;
- *Flux* – obsługujący subskrypcje GraphQL z aplikacji mobilnych.

5.1.2 Routing wiadomości

Przebieg wiadomości zależy od rozmieszczenia rozmówców w sieci. Jeśli obaj są podłączeni do tego samego węzła, wiadomość trafia do odbiorcy lokalnie, bez wychodzenia w sieć Thread. Jeśli natomiast są zarejestrowani na różnych węzłach, pakiet jest przesyłany przez sieć Thread do węzła docelowego, który dostarcza go subskrybującej aplikacji mobilnej. Każdy węzeł rozsyła rozgłoszeniowo informacje o aktywnych użytkownikach, dzięki czemu cała sieć zna aktualny stan obecności uczestników. Oba scenariusze routingu ilustruje rysunek 5.2.



Rysunek 5.2: Scenariusze routingu wiadomości: rozmówcy na różnych węzłach (a) oraz na tym samym węźle (b). Źródło: Opracowanie własne

System nie korzysta z żadnej trwałej bazy danych po stronie infrastruktury. Dane o sesjach, kluczach powitalnych i obecności użytkowników przechowywane są wyłącznie w pamięci operacyjnej węzłów. Wyłączenie węzła usuwa te dane bezpowrotnie. Jest to celowa decyzja projektowa minimalizująca ślad cyfrowy systemu.

5.1.3 Klienci mobilni i model bezpieczeństwa

Klientami systemu są natywne aplikacje na platformę Android, napisane w języku Kotlin. Bezpieczeństwo komunikacji realizowane jest na dwóch niezależnych poziomach.

Na poziomie transportu wymiana kluczy opiera się na protokole PQX3DH (*Post-Quantum Extended Triple Diffie-Hellman*). Węzeł przechowuje wyłącznie klucz powitalny potrzebny do zainicjowania sesji. Po uzgodnieniu klucza komunikacja przechodzi na protokół podwójnej zapadki, zapewniający szyfrowa-

nie od końca do końca z właściwościami utajniania z wyprzedzeniem oraz zapewniająca bezpieczeństwo po naruszeniu. Ewentualne przejście klucza w danej chwili nie pozwala odczytać ani wcześniejszych, ani późniejszych wiadomości. Backend nie posiada dostępu do treści rozmów na żadnym etapie komunikacji.

Na poziomie przechowywania historia rozmów zapisana jest lokalnie w bazie Room, zaszyfrowanej kluczem głównym. Klucz ten przechowywany jest w Magazynie kluczy Android i wymaga uwierzytelnienia biometrycznego do odblokowania. Fizyczny dostęp do urządzenia nie jest wystarczający do odczytania wiadomości.

W rezultacie żaden pojedynczy komponent systemu nie dysponuje wystarczającymi informacjami do odtworzenia treści prowadzonych rozmów.

5.2 Architektura zdecentralizowana

W systemie nie wyróżniono żadnego centralnego serwera. Każdy węzeł jest równorzędnym uczestnikiem sieci, który jednocześnie routuje pakiety Thread, hostuje backend i przyjmuje połączenia od klientów mobilnych. Podczas prac testowano konfiguracje dwu i trójwęzłowe, jednak kod pisany był z założeniem skalowalności do większej liczby węzłów bez modyfikacji architektury.

5.2.1 Role węzłów w sieci Thread

Standard Thread przydziela role węzłom automatycznie podczas formowania sieci. Każdy Raspberry Pi uruchamia demona OTBR, który umożliwia nowym urządzeniom dołączenie do sieci i uczestniczy w routingu pakietów. Jeden z aktywnych routerów wybierany jest automatycznie jako *Leader*, odpowiedzialny za zarządzanie topologią. W przypadku jego awarii protokół przeprowadza nową selekcję spośród pozostałych routerów bez udziału administratora [12].

Przynależność do sieci określa współdzielony klucz sieciowy, podawany przy konfiguracji każdego węzła. Węzeł z poprawnym kluczem łączy się automatycznie i od razu uczestniczy w routingu. Nie jest wymagana ręczna aktualizacja tablic routingu ani centralna rejestracja węzła.

5.2.2 Topologia testowa

Środowisko testowe składało się z dwóch węzłów głównych: Raspberry Pi 4 i Raspberry Pi 5, do których podłączali się klienci mobilni oraz węzeł pomocniczego opartego na Raspberry Pi Zero 2W. Węzeł pomocniczy nie obsługiwał połączeń klientów ze względu na brak interfejsu Ethernet. Nie mógł jednocześnie pełnić funkcji punktu dostępowego Wi-Fi i pozostawać dostępny diagnostycznie. Pełnił natomiast rolę routera pośredniczącego w sieci Thread i był widoczny dla warstwy aplikacyjnej jako pełnoprawny węzeł. Jego obecność potwierdziła poprawne działanie architektury w konfiguracji z więcej niż dwoma węzłami.

5.2.3 Odporność na awarie

Brak centralnego serwera sprawia, że awaria jednego węzła nie wstrzymuje pracy pozostałych. Użytkownicy podłączeni do niedostępnego węzła tracą połączenie, jednak pozostałe węzły działają bez zakłóceń. Wiadomości kierowane do niedostępnego użytkownika są buforowane w pamięci operacyjnej węzła docelowego i dostarczane po ponownym nawiązaniu połączenia. Reset węzła powoduje utratę nieodebranych wiadomości, ponieważ system nie korzysta z trwałego magazynu danych. Przyjęty kompromis omówiono szerzej w rozdziale 7.2.

Utrata węzła pełniącego rolę *Leadera* nie przerywa komunikacji. Protokół Thread automatycznie wybiera nowego *Leadera* spośród aktywnych routerów i ruch sieciowy jest przywracany bez interwencji operatora.

5.3 Warstwa komunikacyjna (OTCTL, Java)

Warstwa komunikacyjna to najniższy poziom stosu aplikacyjnego. Jej zadaniem jest udostępnienie operacji sieci Thread jako metod Javy tak, aby wyższe warstwy nie musiały obsługiwać bezpośrednio demona

ot-daemon ani narzędzia ot-ctl. Składa się z trzech komponentów: klasy OtCtl, klasy NetService oraz klasy ThreadFluxBridge.

5.3.1 Sterowanie siecią Thread — OtCtl

Klasa OtCtl pełni rolę sterownika dla wszystkich operacji na stosie Thread. Każda operacja sieciowa taka jak uruchomienie interfejsu, dołączenie do sieci, odczyt stanu węzła czy komisjonowanie nowych uczestników realizowana jest jako wywołanie narzędzia ot-ctl w postaci osobnego procesu systemowego. Wynik komendy odczytywany jest ze strumienia wyjściowego tego procesu. Szczegóły implementacyjne zawiera rozdział 6.2 oraz 6.3.

Poza opakowaniem komend OtCtl zawiera logikę wyższego poziomu: parsowanie tablicy routerów Thread do obiektów Javy, wyznaczanie adresów RLOC IPv6 z identyfikatora RLOC16 i prefiksu *mesh-local* oraz obsługę awaryjnego czyszczenia konfiguracji sieci. Operacje te wykorzystywane są przez ThreadFluxBridge przy śledzeniu topologii i adresowaniu zdalnych węzłów.

5.3.2 Transport UDP — NetService

Węzły komunikują się za pośrednictwem gniazd UDP IPv6 zarządzanych przez klasę NetService. Wyróżniono dwa kanały komunikacyjne:

- gniazdo multicastowe na adresie `ff03::1` (*mesh-local all-nodes*) – przeznaczone do rozgłoszeń i wykrywania węzłów;
- gniazdo unicastowe – służące do bezpośredniej wymiany pakietów z konkretnym węzłem.

Do adresowania służą adresy RLOC (*Routing Locator*), wyznaczane na podstawie identyfikatora RLOC16 i prefiksu *mesh-local* z aktywnego datasetu Thread.

5.3.3 Most między Thread a Spring — ThreadFluxBridge

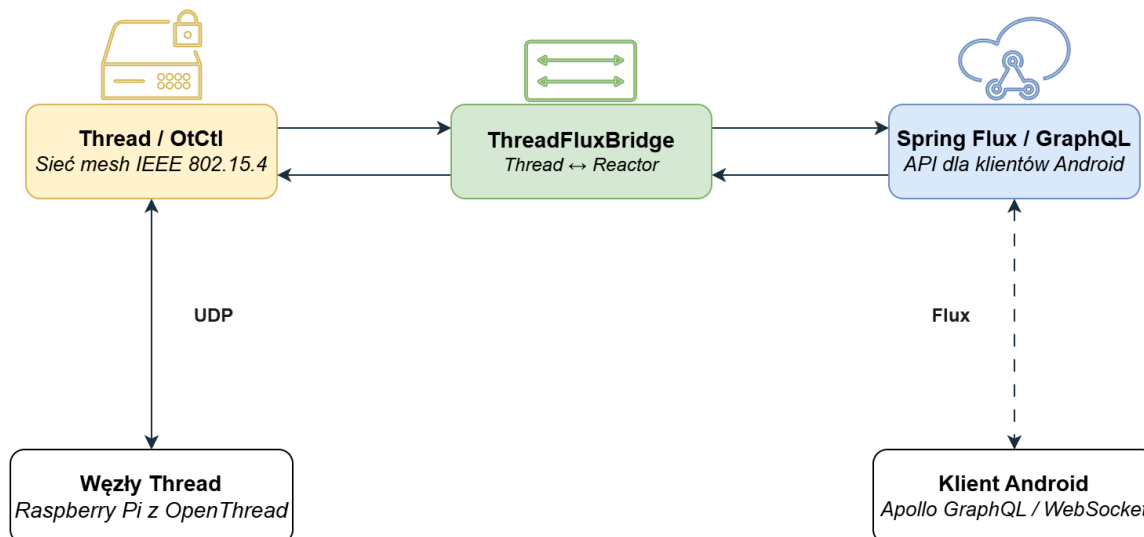
Klasa ThreadFluxBridge łączy warstwę transportową z warstwą aplikacyjną Spring Flux. Tłumaczy pakiety UDP odbierane z sieci Thread na obiekty domenowe Javy i kieruje je do właściwych odbiorców lokalnych klientów bieżącego węzła lub klientów zarejestrowanych na węzłach zdalnych.

W pamięci operacyjnej utrzymywane są dwie kolekcje: zbiór lokalnych klientów bieżącego węzła oraz zbiór klientów zdalnych. Adresy RLOC węzłów zdalnych odświeżane są co 30 sekund na podstawie tablicy routerów odczytywanej z OtCtl, gdyż mogą ulegać zmianie przy rekonfiguracji topologii sieci. Do identyfikacji węzłów służy stabilny adres sprzętowy MAC EUI64, niezależny od aktualnego adresu RLOC.

Ramki przesyłane między węzłami mają postać dokumentów JSON z polem `type` określającym ich semantykę. Obejmują one przesyłanie wiadomości użytkownika, rejestrację i wyrejestrowanie klientów, wzajemne odkrywanie węzłów, publikację kluczy kryptograficznych X3DH oraz synchronizację czasu sieci rozproszonej. Wiadomości przeznaczone dla lokalnych klientów przekazywane są do reaktywnego strumienia, z którego korzystają resolwery GraphQL (*Resolver*) obsługujące połączenia z aplikacjami mobilnymi. Szczegółowa implementacja opisana jest w rozdziale 6.3.3.

5.4 Warstwa serwera (Spring Boot, GraphQL)

Warstwa serwera odpowiada za obsługę połączeń z aplikacjami mobilnymi, zarządzanie użytkownikami i ich kluczami kryptograficznymi oraz routing wiadomości do lokalnych i zdalnych odbiorców. Zrealizowana jest jako aplikacja Spring Boot z interfejsem GraphQL. Kolejne podrozdziały omawiają schemat GraphQL, logikę rozwiązywania zapytań oraz mechanizmy zarządzania aktywnością użytkowników i buforowania wiadomości. Architekturę warstwową węzła oraz przepływ danych między komponentami przedstawia rysunek 5.3.



Rysunek 5.3: Schemat warstw backendu węzła. Źródło: Opracowanie własne

5.4.1 Schemat GraphQL

Interfejs komunikacyjny serwera zdefiniowany jest jako schemat GraphQL obejmujący cztery typy danych oraz trzy kategorie operacji.

Podstawowym typem jest `User`, reprezentujący uczestnika sieci. Zawiera identyfikator, nazwę wyświetlaną, klucze publiczne PQX3DH (`PublicKeys`), flagę aktywności oraz flagę widoczności (*incognito*). Typ `Message` przechowuje zaszyfrowaną wiadomość wraz z nagłówkiem protokołu podwójnej zapadki (`ratchetHeader`), wektorem inicjalizacji (`iv`), numerem wiadomości w sekwencji (`messageNumber`). W pierwszej wiadomości sesji, inicjującej wymianę kluczy PQX3DH, typ ten zawiera dodatkowo klucz efemeryczny nadawcy (`ephemeralKey`) oraz szyfrogram mechanizmu Kyber KEM (`kyberCiphertext`), niezbędny odbiorcy do dekapsulacji wspólnego sekretu. W kolejnych wiadomościach pola te przyjmują wartość `null`. Żadne pole tego typu nie zawiera treści w postaci jawnej – serwer operuje wyłącznie na danych zaszyfrowanych po stronie klienta.

Operacje schematu podzielone są na trzy grupy:

- *Query* — zapytania o dane: pobranie użytkownika, jego kluczy publicznych, listy wiadomości oraz list użytkowników aktywnych i nieaktywnych;
- *Mutation* — operacje modyfikujące stan: publikacja kluczy publicznych PQX3DH (`publishKeys`) oraz wysłanie wiadomości (`sendMessage`), które w pierwszej wiadomości sesji przyjmuje dodatkowo klucz efemeryczny nadawcy (`ephemeralKey`) oraz szyfrogram mechanizmu Kyber KEM (`kyberCiphertext`);
- *Subscription* — subskrypcja reaktywna `onNewMessage`, przez którą klient mobilny odbiera wiadomości w czasie rzeczywistym.

5.4.2 Resolwery GraphQL

Logika obsługi operacji GraphQL podzielona jest między dwa resolwery: `UserResolver` i `MessageResolver`.

`UserResolver` obsługuje operacje związane z użytkownikami. Mutacja `publishKeys` rejestruje użytkownika w systemie: zapisuje jego klucze publiczne PQX3DH lokalnie w serwisie `UserService`, rejestruje klienta w `ThreadFluxBridge` i rozsyła klucze rozgłoszeniowo do pozostałych węzłów sieci. Operacja ta inicjuje obecność użytkownika w całej sieci rozproszonej. Flaga `isVisible` pozwala użytkownikowi działać w trybie incognito – jest on wówczas niewidoczny na listach użytkowników innych klientów, ale nadal może uczestniczyć w komunikacji.

`MessageResolver` obsługuje wysyłanie i odbieranie wiadomości. Mutacja `sendMessage` wyznacza trasę wiadomości na podstawie lokalizacji odbiorcy: jeśli odbiorca jest klientem lokalnym z aktywną subskrypcją, wiadomość jest dostarczana natychmiast przez reaktywny strumień; jeśli jest klientem zdalnym, jest przekazywana do `ThreadFluxBridge` i wysyłana przez sieć Thread; jeśli odbiorca jest chwilowo niedostępny, wiadomość trafia do bufora `MessageCache`. Subskrypcja `onNewMessage` łączy dwa strumienie reaktywne: strumień z `MessageService` dla wiadomości lokalnych oraz strumień z `ThreadFluxBridge` dla wiadomości przychodzących z sieci Thread. Po nawiązaniu subskrypcji użytkownik automatycznie otrzymuje wiadomości zbuforowane podczas jego nieobecności.

5.4.3 Zarządzanie aktywnością użytkowników

Aktywność użytkowników śledzona jest przez serwis `UserActivityManager`. Dla każdego użytkownika przechowywany jest znacznik czasu ostatniej aktywności. Użytkownik uznawany jest za aktywnego, jeśli od jego ostatniej akcji nie upłynęło więcej niż 30 sekund. Każda wysłana wiadomość oraz każda aktywna subskrypcja odświeżają ten znacznik.

Cykliczne zadanie uruchamiane co 10 sekund weryfikuje stan wszystkich zarejestrowanych użytkowników. W przypadku wykrycia zmiany statusu (aktywny → nieaktywny lub odwrotnie) zmiana jest rozsyłana rozgłoszeniowo do pozostałych węzłów sieci przez `ThreadFluxBridge`, dzięki czemu każdy węzeł posiada aktualną informację o dostępności użytkowników w całej sieci rozproszonej. Informacje te przechowywane są wyłącznie w pamięci operacyjnej – reset węzła usuwa je bezpowrotnie.

5.4.4 Buforowanie wiadomości

Serwis `MessageCache` buforuje wiadomości kierowane do użytkowników chwilowo niedostępnych. Dla każdego użytkownika utrzymywana jest kolejka oczekujących wiadomości z limitem 100 pozycji – po jego przekroczeniu najstarsza wiadomość jest usuwana. Każda wiadomość przechowywana jest wraz ze znacznikiem czasu dodania. Cykliczne zadanie uruchamiane co godzinę usuwa wiadomości starsze niż 24 godziny.

Po ponownym nawiązaniu subskrypcji `onNewMessage` przez klienta wszystkie zbuforowane wiadomości są pobierane z kolejki i dostarczane przed strumieniem nowych wiadomości, co gwarantuje zachowanie kolejności i kompletności historii rozmów po przerwie w połączeniu.

5.5 Aplikacja mobilna (Android, Kotlin)

Aplikacja mobilna to jedyna warstwa systemu widoczna bezpośrednio dla użytkownika. Napisana jest w języku Kotlin na platformę Android, z interfejsem opartym na Jetpack Compose. Architekturę aplikacji przedstawia rysunek 5.4. Poniższe podrozdziały omawiają poszczególne warstwy: komunikację z węzłem, lokalne przechowywanie danych oraz moduł kryptograficzny.

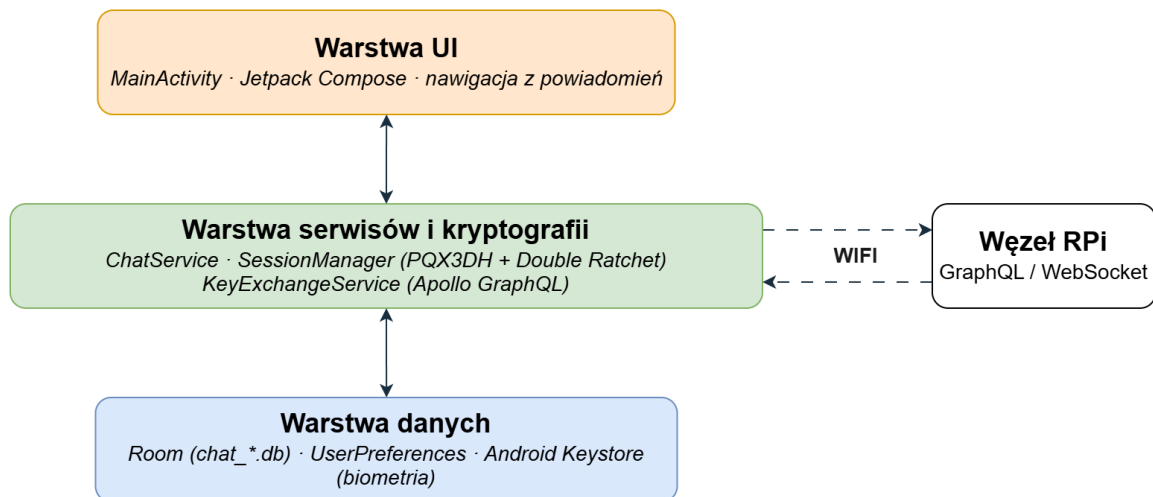
5.5.1 Architektura i stos technologiczny

Interfejs użytkownika zbudowany jest w całości w Jetpack Compose, z nawigacją opartą na bibliotece `androidx.navigation.compose`. Punktem wejścia jest klasa `MainActivity`, rozszerzająca `FragmentActivity`. Takie dziedziczenie jest wymagane przez `BiometricPrompt` do uwierzytelnienia biometrycznego. Nawigacja obsługuje przekierowania z powiadomień systemowych - kliknięcie powiadomienia o nowej wiadomości przenosi użytkownika bezpośrednio do okna rozmowy z nadawcą.

W tle działa usługa pierwszoplanowa `ChatService`, zadeklarowana z typem `dataSync`. Utrzymuje ona aktywne połączenie z węzłem i obsługuje subskrypcję GraphQL `onNewMessage`, nawet gdy aplikacja nie jest widoczna na pierwszym planie. Dzięki temu wiadomości i powiadomienia docierają do użytkownika niezależnie od stanu cyklu życia aktywności.

5.5.2 Komunikacja z węzłem — Apollo GraphQL

Komunikacja z backendowym węzłem odbywa się przez bibliotekę Apollo Kotlin, będącą klientem GraphQL generującym kod na podstawie schematu serwera. Apollo obsługuje wszystkie trzy typy operacji:



Rysunek 5.4: Uproszczony schemat architektury aplikacji Android. Źródło: Opracowanie własne

zapytania (*query*) pobierające dane o użytkownikach i kluczach, mutacje (*mutation*) do publikacji kluczy PQX3DH i wysyłania wiadomości oraz subskrypcje (*subscription*) dostarczające wiadomości w czasie rzeczywistym przez WebSocket. Aplikacja łączy się z węzłem przez adres IP w sieci lokalnej Wi-Fi, do której węzeł Raspberry Pi udostępnia punkt dostępowy.

5.5.3 Lokalne przechowywanie danych — Room

Historia rozmów przechowywana jest lokalnie w bazie Room z szyfrowaniem na poziomie warstwy aplikacyjnej. Klucz główny jest przechowywany w Magazynie kluczy Android i chroniony jest uwierzytelnieniem biometrycznym. Dostęp do klucza wymaga potwierdzenia tożsamości przez odcisk palca lub rozpoznanie twarzy. Dodatkową warstwę ochrony stanowi `EncryptedSharedPreferences` z biblioteki `androidx.security.crypto`, służące do przechowywania konfiguracji i stanu sesji kryptograficznej. Fizyczny dostęp do urządzenia nie wystarczy do odczytania historii rozmów bez uwierzytelnienia biometrycznego.

5.5.4 Moduł kryptograficzny — PQX3DH i podwójna zapadka

Cała logika kryptograficzna realizowana jest po stronie klienta. Do operacji niskopoziomowych wykorzystywana jest biblioteka Bouncy Castle (`bcprov-jdk18on`), zapewniająca implementację krzywej X25519, mechanizmu Kyber KEM oraz funkcji wyprowadzania kluczy. Uzgodnienie klucza sesji realizuje protokół PQX3DH zaimplementowany w klasie `PQX3DHProtocol`. Obliczane są cztery niezależne sekrety X25519 (DH1–DH4), a inicjator dodatkowo enkapsuluje klucz publiczny Kyber-768 odbiorcy, uzyskując szyfrogram (`kyberCiphertext`) oraz wspólny sekret KEM. Odbiorca dekapsuluje szyfrogram swoim kluczem prywatnym Kyber, odtwarzając identyczny sekret. Wszystkie składniki – DH1, DH2, DH3, DH4 oraz sekret Kyber KEM – konkatenują się w stałej kolejności i przepuszczane przez funkcję KDF z etykietą `PQX3DH_SESSION_KEY_v2`. Stała kolejność składników zapewnia symetryczność protokołu – obie strony uzyskują identyczny klucz sesji niezależnie od kolejności obliczeń. Włączenie sekretu Kyber do wyprowadzania klucza sprawia, że bezpieczeństwo sesji opiera się zarówno na trudności problemu Diffiego–Hellmana na krzywych eliptycznych, jak i na odporności Kyber na ataki komputerów kwantowych. Po uzgodnieniu klucza sesji dalsza komunikacja prowadzona jest protokołem podwójnej zapadki, zapewniającym utajnienie z wyprzedzeniem i bezpieczeństwo po naruszeniu. Szczegółowa implementacja protokołów opisana jest w rozdziale 6.4.2.

5.6 Model danych i przechowywanie informacji (Room)

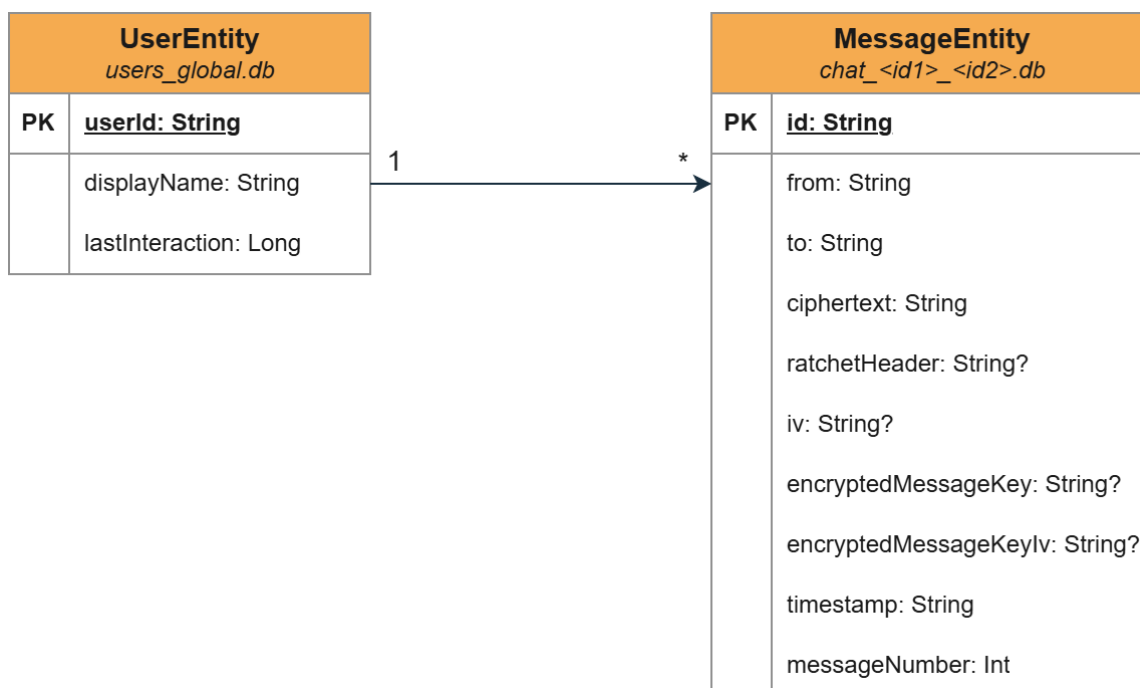
Dane aplikacji przechowywane są lokalnie na urządzeniu z użyciem biblioteki Room – warstwy abstrakcji nad SQLite. Zastosowano dwie niezależne bazy danych, różniące się zakresem danych i cyklem życia. Poniższe podrozdziały opisują encje danych, strategię izolacji rozmów oraz warstwę dostępu do danych.

5.6.1 Encje danych

Model danych składa się z dwóch encji: `UserEntity` i `MessageEntity`, których strukturę przedstawia rysunek 5.5.

`UserEntity` przechowuje informacje o znanych uczestnikach sieci: identyfikator, nazwę wyświetlaną i znacznik czasu ostatniej interakcji. Dane te są trwałe – przeżywają restart aplikacji i są dostępne bez połączenia z węzłem.

`MessageEntity` przechowuje wiadomości z pełną strukturą kryptograficzną wymaganą przez protokół podwójnej zapadki. Każda wiadomość zawiera zaszyfrowaną treść (`ciphertext`), nagłówek protokołu (`ratchetHeader`) w formacie `base64PublicKey:msgNum:prevChainLen`, wektor inicjalizacji AES-GCM (`iv`) oraz numer wiadomości w łańcuchu (`messageNumber`). Dla wiadomości wysłanych przez bieżącego użytkownika przechowywany jest dodatkowo klucz wiadomości zaszyfrowany przez Magazyn kluczy Android (`encryptedMessageKey` i `encryptedMessageKeyIv`). Umożliwia to ponowne odszyfrowanie własnych wiadomości po uwierzytelnieniu biometrycznym. Dla wiadomości przychodzących pola te są puste – klucz odtwarzany jest ze stanu podwójnej zapadki.



Rysunek 5.5: Diagram encji modelu danych aplikacji Android. Źródło: Opracowanie własne

5.6.2 Izolacja rozmów

Każda para użytkowników ma odrębną bazę danych SQLite o nazwie `chat_...db`. Identyfikatory są leksykograficznie posortowane, co gwarantuje tę samą nazwę niezależnie od tego, która ze stron inicjuje rozmowę. Tworzeniem i zarządzaniem instancjami zajmuje się klasa `DatabaseProvider`.

Taka strategia ma dwie zalety. Usunięcie rozmowy sprowadza się do usunięcia jednego pliku, bez ryzyka naruszenia innych rozmów. Wykrywanie istniejących konwersacji odbywa się przez listowanie

plików w katalogu baz danych i filtrowanie wzorcem `chat_*.db` – nie potrzeba centralnej tabeli katalogu.

Odrębna baza `users_global.db` przechowuje encje `UserEntity` współdzielone między rozmowami. Rozdzielenie to upraszcza zarządzanie cyklem życia danych: usunięcie rozmowy nie usuwa informacji o użytkowniku z listy kontaktów.

5.6.3 Warstwa dostępu do danych

Dostęp do baz danych realizowany jest przez Interfejsy warstwy danych: `MessageDao` i `UserDao`. Zapytania zwracają obiekty reaktywne `Flow` z biblioteki Kotlin Coroutines. Dzięki temu interfejs automatycznie reaguje na zmiany danych bez jawnego odpytywania bazy. Warstwę pośredniczącą między DAO a pozostałymi komponentami stanowią `UserRepository` i `ChatDatabaseManager`, ukrywające szczegóły wyboru właściwej instancji bazy dla danej pary użytkowników.

5.7 Mechanizmy bezpieczeństwa i szyfrowania (PQX3DH)

Bezpieczeństwo systemu opiera się na trzech niezależnych warstwach kryptograficznych: uzgodnieniu klucza sesji przez protokół PQX3DH, szyfrowaniu wiadomości przez protokół podwójnej zapadki oraz ochronie kluczy przez Magazyn kluczy Android z uwierzytelnieniem biometrycznym. Poniższe podrozdziały omawiają każdą z tych warstw.

5.7.1 Uzgodnienie klucza sesji — PQX3DH

Nawiązanie sesji kryptograficznej realizowane jest przez protokół PQX3DH, zaimplementowany w klasie `PQX3DHProtocol`. Jest to rozszerzenie standardowego protokołu X3DH [15] o dodatkową wymianę kluczy z wykorzystaniem mechanizmu Kyber KEM, zwiększającą odporność na ataki komputerów kwantowych.

W trakcie uzgodnienia obliczane są cztery niezależne sekrety Diffie-Hellman na krzywej X25519:

- DH1 – między kluczem tożsamości inicjatora a kluczem podpisanym odbiorcy;
- DH2 – między efemerycznym kluczem inicjatora a kluczem tożsamości odbiorcy;
- DH3 – między efemerycznym kluczem inicjatora a kluczem podpisanym odbiorcy;
- DH4 – między efemerycznym kluczem inicjatora a kluczem jednorazowym (*one-time pre-key*) odbiorcy, o ile jest dostępny; w przeciwnym razie stosowany jest zerowy wektor 32 bajtów.

Inicjator enkapsuluje ponadto klucz publiczny Kyber-768 odbiorcy, uzyskując szyfrogram oraz dodatkowy sekret KEM. Odbiorca dekapuluje szyfrogram swoim kluczem prywatnym Kyber, odtwarzając identyczny sekret. Wszystkie składniki – DH1, DH2, DH3, DH4 oraz sekret Kyber KEM – konkatelowane są w stałej kolejności i przepuszczane przez funkcję HKDF (*HMAC-based Key Derivation Function*) z etykietą `PQX3DH_SESSION_KEY_v2`. Wynikiem jest 32-bajtowy klucz sesji.

Stała kolejność składników zapewnia symetryczność protokołu - obie strony uzyskują identyczny klucz niezależnie od kolejności obliczeń. Hybrydowe połączenie X25519 i Kyber-768 gwarantuje bezpieczeństwo nawet w przypadku przełomu w kryptoanalizie któregoś z tych algorytmów – skompromitowanie jednego nie wystarcza do odtworzenia klucza sesji.

Pełny przebieg uzgodnienia, obejmujący rejestrację kluczy publicznych na serwerze, pobranie kluczy drugiej strony oraz lokalną derywację klucza sesji, przedstawia rysunek 5.6. Implementacja kryptograficzna opiera się na bibliotece Bouncy Castle z obsługą krzywej X25519, algorytmu Kyber-768 i funkcji HMAC-SHA256.

Węzeł uczestniczy w uzgodnieniu wyłącznie jako repozytorium kluczy publicznych. Przechowuje klucz tożsamości, klucz podpisany, klucze jednorazowe oraz klucz publiczny Kyber każdego użytkownika, publikowane przez mutację `publ` i `shKeys`. Po zakończeniu uzgodnienia węzeł nie posiada żadnych informacji pozwalających odtworzyć klucz sesji.

5.7.2 Szyfrowanie wiadomości — podwójna zapadka

Po uzgodnieniu klucza sesji komunikacja prowadzona jest protokołem podwójnej zapadki [16]. Każda wiadomość szyfrowana jest algorytmem AES-GCM z unikalnym 256-bitowym kluczem, wyprowadzanym z bieżącego stanu łańcucha kluczy. Razem z wiadomością przesyłany jest nagłówek zawierający klucz publiczny DH nadawcy, numer wiadomości w łańcuchu i długość poprzedniego łańcucha, w postaci `base64PublicKey:msgNum:prevChainLen`.

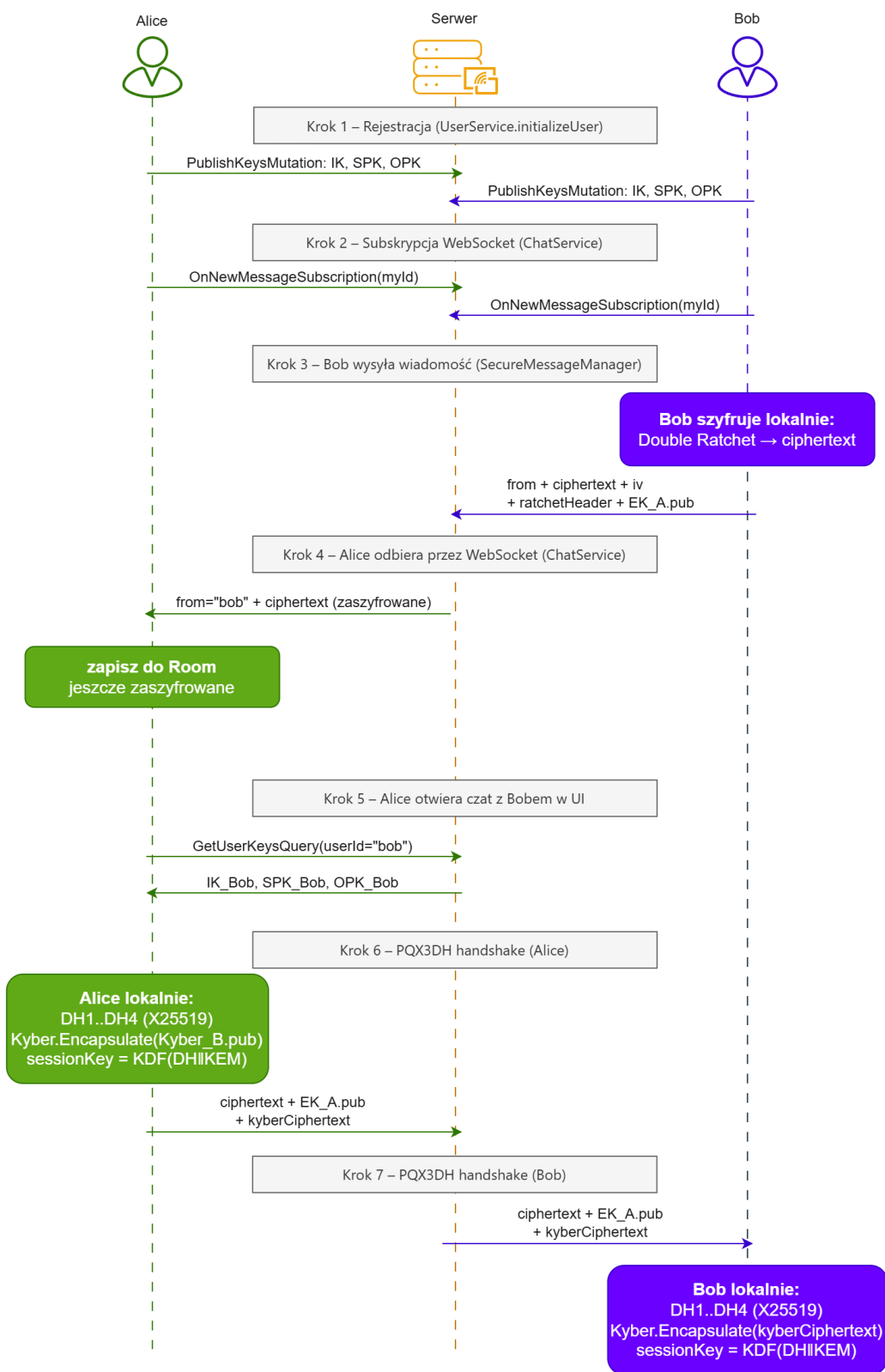
Protokół zapewnia dwie kluczowe właściwości. Doskonała poufność przekazu gwarantuje, że ujawnienie bieżącego klucza nie pozwala odszyfrować wcześniejszych wiadomości – klucze są wyprowadzane jednokierunkowo i niszczone po użyciu. Bezpieczeństwo po naruszeniu sprawia, że po ewentualnym przejęciu stanu protokołu kolejna wymiana kluczy DH przywraca poufność komunikacji.

5.7.3 Ochrona kluczy — Magazyn kluczy Android i biometria

Klucze kryptograficzne przechowywane są w Magazynie kluczy Android, izolowanym magazynie niedostępnym dla warstwy aplikacyjnej w postaci jawnej. Klucz główny o aliasie `MessageKeyMaster` generowany jest algorytmem AES-256-GCM z wymaganym uwierzytelnieniem użytkownika. Skonfigurowany jest z oknem ważności 600 sekund - po jego upływie każda operacja kryptograficzna wymaga ponownego potwierdzenia tożsamości przez silną biometrię lub poświadczenie urządzenia.

Klucz główny służy do szyfrowania kluczy wiadomości wysłanych przez bieżącego użytkownika. Zasyfrowane klucze przechowywane są w bazie Room obok zasyfrowanej treści. Umożliwia to odczytanie własnych wiadomości po uwierzytelnieniu biometrycznym. Dla wiadomości przychodzących klucze nie są persystowane – odtwarzane są ze stanu podwójnej zapadki podczas odczytu.

Uwierzytelnieniem biometrycznym zarządza klasa `BiometricAuthManager`, opakowująca asynchroniczne API `BiometricPrompt` w korutynę Kotlin. Obsługiwane są trzy metody: odcisk palca, rozpoznanie twarzy oraz PIN urządzenia jako mechanizm zastępczy.



Rysunek 5.6: Diagram uzgodnienia klucza sesji w protokole PQX3DH. Źródło: Opracowanie własne

Rozdział 6

Implementacja systemu

Niniejszy rozdział przedstawia implementację zdecentralizowanego komunikatora opisanego w rozdziale poprzednim, obejmującą zarówno warstwę backendową działającą na węzłach Raspberry Pi, jak i natywną aplikację mobilną na platformę Android. Omówiono środowisko implementacyjne, proces konfiguracji sieci Thread, szczegóły implementacyjne poszczególnych komponentów systemu oraz problemy napotkane w trakcie realizacji projektu wraz z zastosowanymi rozwiązaniami. Podrozdział 6.1 opisuje środowisko sprzętowe oraz zależności projektowe backendu i aplikacji mobilnej, a podrozdział 6.2 przedstawia konfigurację OpenThread i OTBR, w tym budowanie narzędzi, uruchamianie demona oraz konfigurację punktu dostępowego Wi-Fi. Podrozdziały 6.3–6.4 omawiają implementację backendu – warstwę Spring WebFlux, transport Thread oraz most ThreadFluxBridge – a także implementację aplikacji mobilnej, obejmującą warstwę Room, protokoły PQX3DH i podwójnej zapadki oraz interfejs użytkownika. Podrozdział 6.5 przedstawia implementację warstwy GraphQL po stronie klienta, natomiast podrozdział 6.6 zamyka rozdział opisem najistotniejszych problemów implementacyjnych i sposobów ich rozwiązania.

6.1 Środowisko implementacyjne

System składa się z dwóch niezależnie rozwijanych komponentów: backendowej aplikacji serwerowej działającej na węzłach Raspberry Pi oraz natywnej aplikacji mobilnej na platformę Android. Każdy z komponentów powstawał w odrębnym środowisku deweloperskim i wykorzystuje osobny łańcuch narzędzi budowania.

6.1.1 Środowisko sprzętowe

Węzły sieci oparto na mikrokomputerach Raspberry Pi 4 i Raspberry Pi 5 z systemem Raspbian Lite 13 (Trixie) w wariantcie 64-bitowym na architekturę ARM, bez środowiska graficznego. Minimalna instalacja systemu ogranicza zużycie pamięci operacyjnej i zmniejsza powierzchnię ataku. Rolę modułu radiowego pełni koordynator USB Sonoff Zigbee 3.0 Dongle Plus-E oparty na układzie Silicon Labs EFR32MG21. Moduł dostarczany jest z fabrycznie zainstalowanym oprogramowaniem Zigbee i wymaga przeprogramowania na oprogramowanie układowe zgodne z OpenThread w trybie RCP (*Radio Co-Processor*).

Przeprogramowania dokonano przy użyciu webowego narzędzia dostępnego pod adresem `dongle.sonoff.tech/sonoff-dongle-flasher`, działającego w przeglądarce z dostępem do portu szeregowego przez *Web Serial API*. Komunikacja z układem EFR32MG21 odbywa się za pośrednictwem konwertera Silicon Labs CP2102, dla którego wymagana jest instalacja dedykowanego sterownika. Po wyborze obrazu oprogramowania Thread RCP i nawiązaniu połączenia z urządzeniem narzędzie samodzielnie przeprowadza procedurę programowania.

6.1.2 Backend

Aplikacja backendowa napisana jest w języku Java 17 z użyciem frameworka Spring Boot 3.2.0 [25]. Projekt zarządzany jest narzędziem Maven; artefaktem wynikowym jest wykonywalny plik JAR zawierający wszystkie zależności, co upraszcza wdrożenie na węzłach bez konieczności instalowania dodatkowych bibliotek w środowisku docelowym. Kod rozwijano w środowisku IntelliJ IDEA na systemach Windows oraz macOS (ARM). Główne zależności projektu backendowego przedstawia tabela 6.1.

6.1.3 Aplikacja mobilna

Aplikacja mobilna napisana jest w języku Kotlin 2.0.21 na platformę Android. Projekt zarządzany jest narzędziem Gradle z wtyczką KSP (*Kotlin Symbol Processing*) do generowania kodu w czasie kompilacji, wykorzystywaną przede wszystkim przez bibliotekę Room [20]. Minimalną obsługiwaną wersją systemu

Tabela 6.1: Główne zależności projektu backendowego

Biblioteka	Wersja	Przeznaczenie
Spring Boot [25]	3.2.0	framework aplikacyjny
Spring WebFlux [26]	(Boot BOM)	reaktywny stos HTTP/WebSocket
Spring GraphQL	(Boot BOM)	warstwa GraphQL
Reactor Core	(Boot BOM)	strumienie reaktywne
Jackson Databind	(Boot BOM)	serializacja JSON
graphql-java-extended-scalars	21.0	dodatkowe skalary GraphQL
Lombok	(Boot BOM)	generowanie kodu pomocniczego

jest Android 14 (`minSdk = 34`), a docelową i kompilacyjną Android 15 (`targetSdk/compileSdk = 36`). Wysoki próg minimalnej wersji systemu wynika z zastosowania interfejsów API wprowadzonych w Androidzie 14, w szczególności nowych mechanizmów uwierzytelniania biometrycznego. Komunikacja z węzłem realizowana jest przez bibliotekę Apollo Kotlin [31]. Interfejs użytkownika zbudowany jest w całości w Jetpack Compose. Kod rozwijano w Android Studio na systemach Windows i macOS (ARM). Główne zależności projektu mobilnego zestawiono w tabeli 6.2.

Tabela 6.2: Główne zależności projektu aplikacji mobilnej

Biblioteka	Wersja	Przeznaczenie
Kotlin	2.0.21	język implementacji
Apollo Kotlin [31]	4.3.3	klient GraphQL / WebSocket
Room KTX [20]	2.8.2	lokalna baza danych SQLite
Bouncy Castle [32]	1.82	kryptografia (X25519, HKDF)
androidx.security-crypto	1.1.0-alpha07	<i>EncryptedSharedPreferences</i>
DataStore Preferences	1.1.7	przechowywanie konfiguracji
Jetpack Compose BOM	2024.09.00	interfejs użytkownika
Navigation Compose	2.9.5	nawigacja między ekranami
Lifecycle Service	2.9.4	usługa pierwszoplanowa

6.2 Konfiguracja OpenThread i OTBR

Niniejszy podrozdział opisuje proces przygotowania węzłów sieci Thread, obejmujący budowanie narzędzi OpenThread, konfigurację demona `ot-daemon` oraz konfigurację punktu dostępowego Wi-Fi na Raspberry Pi.

6.2.1 Budowanie narzędzi OpenThread

Projekt OpenThread nie dostarcza gotowych pakietów binarnych dla platformy Raspberry Pi, dlatego narzędzia `ot-daemon` i `ot-ctl` zbudowano ze źródeł [13]. Próba kompilacji na Raspberry Pi Zero 2W zakończyła się niepowodzeniem ze względu na niewystarczające zasoby pamięciowe – czterokrotne uruchomienie procesu budowania kończyło się błędem braku pamięci, nawet po włączeniu pliku wymiany przez pakiet `dphys-swapfile`. Kompilację przeprowadzono na Raspberry Pi 4 i Raspberry Pi 5, na których przebiegła poprawnie. Listing 6.1 przedstawia sekwencję poleceń wykorzystaną do zbudowania narzędzi.

Listing 6.1: Budowanie narzędzi OpenThread ze źródeł

```

sudo apt update
sudo apt install -y git build-essential cmake ninja-build \
    pkg-config
git clone https://github.com/openthread/openthread.git
cd openthread
./script/bootstrap
./script/cmake-build posix \
    -DOT_DAEMON=ON \
    -DOT_RCP=ON \
    -DOT_COVERAGE=OFF \

```

W wyniku kompilacji powstają dwa pliki wykonywalne: `ot-daemon` oraz `ot-ctl`. Skompilowane binaria przeznaczone dla architektury ARM64 są osadzane bezpośrednio w archiwum JAR aplikacji backendowej w ścieżce `/native/arm64/`. Podczas uruchomienia klasa `NativeBinaries` wypakowuje je do tymczasowego katalogu i nadaje uprawnienia wykonywania, co eliminuje konieczność ręcznej instalacji narzędzi na węźle docelowym. Katalog tymczasowy jest automatycznie usuwany po zakończeniu procesu. Listing 6.2 przedstawia implementację tej klasy.

Listing 6.2: Wypakowanie binarnych zależności OpenThread z archiwum JAR – klasa `NativeBinaries`

```
public static synchronized void extractAll() throws IOException {
    if (tempDir != null) return;
    tempDir = Files.createTempDirectory("openthread-");
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        try { deleteDir(tempDir); } catch (IOException ignored) {}
    }));
    extractBinary("ot-ctl");
    extractBinary("ot-daemon");
}

private static void extractBinary(String name) throws IOException {
    String resource = "/native/arm64/" + name;
    Path target = tempDir.resolve(name);
    try (InputStream in =
        NativeBinaries.class.getResourceAsStream(resource)) {
        if (in == null)
            throw new IOException(
                "Brakuje binarki w JAR: " + resource);
        Files.copy(in, target, StandardCopyOption.REPLACE_EXISTING);
    }
    Set<PosixFilePermission> perms =
        PosixFilePermissions.fromString("rwxr-xr-x");
    Files.setPosixFilePermissions(target, perms);
}
```

Przyjęte rozwiązanie ogranicza się do architektury ARM64, dla której skompilowano osadzone binaria. Architektura JAR przewiduje jednak możliwość rozszerzenia – wystarczy dodać binaria skompilowane dla docelowej platformy (np. `x86_64` lub `armv7`) w odpowiedniej ścieżce wewnątrz archiwum i zaktualizować ścieżkę zasobu w klasie `NativeBinaries`.

6.2.2 Uruchamianie demona Thread

Demon `ot-daemon` uruchamiany jest z parametrami wskazującymi interfejs sieciowy Thread oraz adres modułu radiowego. Klucz `Sonoff` rozpoznawany jest przez system operacyjny automatycznie pod ścieżką `/dev/ttyUSB0`. Polecenie uruchamiające demona przedstawia listing 6.3.

Listing 6.3: Uruchomienie demona `ot-daemon`

```
sudo ot-daemon -I wpan0 \
    'spinel+hdlc+uart:///dev/ttyUSB0?uart-baudrate=460800'
```

Parametr `-I wpan0` definiuje nazwę wirtualnego interfejsu sieciowego Thread tworzonego przez demona. Adres URL modułu radiowego przekazywany jest w formacie protokołu Spinel przez transport HDLC/UART z prędkością 460800 baudów, wymaganą przez firmware Thread RCP układu EFR32MG21.

6.2.3 Zarządzanie siecią Thread z poziomu aplikacji

Sterowanie demonem realizowane jest przez klasę `OtCtl`, która wywołuje narzędzie `ot-ctl` jako zewnętrzny proces systemowy. Każde wywołanie tworzy nowy proces, przekazuje komendę i odczytuje wynik ze strumienia wyjściowego. Listing 6.4 przedstawia centralną metodę wykonującą polecenia.

Listing 6.4: Metoda wykonująca polecenia ot-ctl

```
public static String run(String raw) {
    try {
        Process p = new ProcessBuilder(cmd(raw))
            .redirectErrorStream(true)
            .start();
        p.waitFor();
        try (BufferedReader br = new BufferedReader(
            new InputStreamReader(p.getInputStream()))) {
            return br.lines()
                .collect(Collectors.joining("\n"))
                .trim();
        }
    } catch (Exception e) {
        return "ERR: " + e.getMessage();
    }
}
```

Podczas uruchamiania aplikacja sprawdza, czy aktywny dataset sieci Thread już istnieje. Jeśli nie – inicjuje procedurę dołączenia do istniejącej sieci jako *joiner* z konfigurowalnym kluczem PSKd (*Pre-Shared Key for the Device*). Po upływie limitu czasu (120 sekund) lub na żądanie operatora węzeł tworzy własną sieć i uruchamia komisjonera akceptującego nowe urządzenia. Listing 6.5 przedstawia logikę procedury rozruchowej.

Listing 6.5: Procedura rozruchowa węzła – dołączenie lub utworzenie sieci Thread

```
private static void bootstrapJoinOrCreateWithSkip() {
    if (OtCtl.hasActiveDataset()) {
        System.out.println("[BOOT] Active dataset znaleziony.");
        OtCtl.ensureThreadStarted();
        return;
    }
    //czeka na join przez 120 s lub do naciśnięcia ENTER
    boolean joined = OtCtl.joinerStartAndWaitInterruptible(
        Config.PSK, 120, skip);

    if (joined) {
        OtCtl.ensureThreadStarted();
    } else {
        if (OtCtl.createNetworkAndStart()) {
            OtCtl.commissionerAllowAll(Config.PSK);
        }
    }
}
```

6.2.4 Konfiguracja punktu dostępowego Wi-Fi

Każdy węzeł pełni jednocześnie rolę punktu dostępowego Wi-Fi, przez który łączą się aplikacje mobilne. Konfiguracja opiera się na pakietach `hostapd` i `dnsmasq`. Interfejs `wlan0` jest wyłączony z zarządzania przez `NetworkManager` i konfigurowany statycznie z adresem `192.168.50.1/24`. Serwer DHCP `dnsmasq` przydziela klientom adresy z zakresu `192.168.50.10–192.168.50.100`. IPv6 na interfejsie `wlan0` jest wyłączone, aby ruch sieciowy IPv6 stosu Thread nie przechodził przez interfejs Wi-Fi.

Punkt dostępowy skonfigurowano w paśmie 5 GHz (kanał 36, standard IEEE 802.11ac) z szyfrowaniem WPA2-PSK. Listing 6.6 przedstawia plik konfiguracyjny `hostapd`.

Listing 6.6: Plik konfiguracyjny punktu dostępowego `/etc/hostapd/hostapd.conf`

```
interface=wlan0
driver=nl80211
ssid=Pi_Local_Server
country_code=PL
hw_mode=a
channel=36
ieee80211n=1
ieee80211ac=1
```

```
wmm_enabled=1
wpa=2
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
wpa_passphrase=BezpieczneHaslo123
```

Wartość `hw_mode=a` wskazuje pasmo 5 GHz, a `channel=36` wybrany kanał. Obsługa standardów 802.11n i 802.11ac włączona jest odpowiednio przez flagi `ieee80211n` i `ieee80211ac`. Opcja `wmm_enabled` aktywuje mechanizm priorytetyzacji ruchu *Wi-Fi Multimedia*, wymagany przez standard 802.11ac.

6.2.5 Uruchamianie aplikacji

Aplikacja backendowa uruchamiana jest ręcznie jako plik JAR z parametrami wiersza poleceń. Listing 6.7 przedstawia przykładowe polecenie startowe.

Listing 6.7: Uruchomienie węzła backendowego

```
java -jar thread-chat-1.3.2-jar-with-dependencies.jar \
  --name NAZWA \
  --iface wlan0 \
  --psk J01NME \
  --server
```

Parametr `-name` określa nazwę węzła widoczną w sieci, `-iface` wskazuje interfejs Thread, `-psk` ustala klucz komisjonowania, a `-server` uruchamia warstwę Spring Boot z interfejsem GraphQL. Aplikacja obsługuje dwa tryby uruchomienia. Tryb standardowy bez flagi `-server` uruchamia wyłącznie warstwę transportową Thread z interfejsem wiersza poleceń, co było wykorzystywane na wczesnym etapie prac do testowania komunikacji między węzłami bez warstwy aplikacyjnej. Tryb serwerowy z flagą `-server` uruchamia dodatkowo kontener Spring Boot z interfejsem GraphQL i panelem administracyjnym – jest to tryb docelowy stosowany podczas normalnej eksploatacji węzła.

Pełne zestawienie parametrów uruchomienia przedstawia tabela 6.3.

Tabela 6.3: Parametry uruchomienia węzła backendowego

Parametr	Opis	Przykład
<code>-name</code>	nazwa węzła widoczna w sieci Thread	<code>AP-Living</code>
<code>-iface</code>	nazwa interfejsu sieciowego Thread	<code>wlan0</code>
<code>-psk</code>	klucz komisjonowania PSKd	<code>J01NME</code>
<code>-server</code>	uruchomienie warstwy Spring Boot	(flaga)
<code>-secret-hex</code>	opcjonalny klucz transportowy (32 B)	<code>abc123...</code>

6.2.6 Panel administracyjny

Warstwę administracyjną systemu uzupełnia panel webowy dostępny przez przeglądarkę, zrealizowany z użyciem szablonów Thymeleaf i Spring MVC. Kontroler panelu (`AdminController`) obsługuje zapytania pod ścieżką `/admin` i udostępnia dwie warstwy interfejsu: widoki HTML renderowane przez Thymeleaf oraz punkty końcowe REST zwracające dane w formacie JSON, odczytywane przez JavaScript strony.

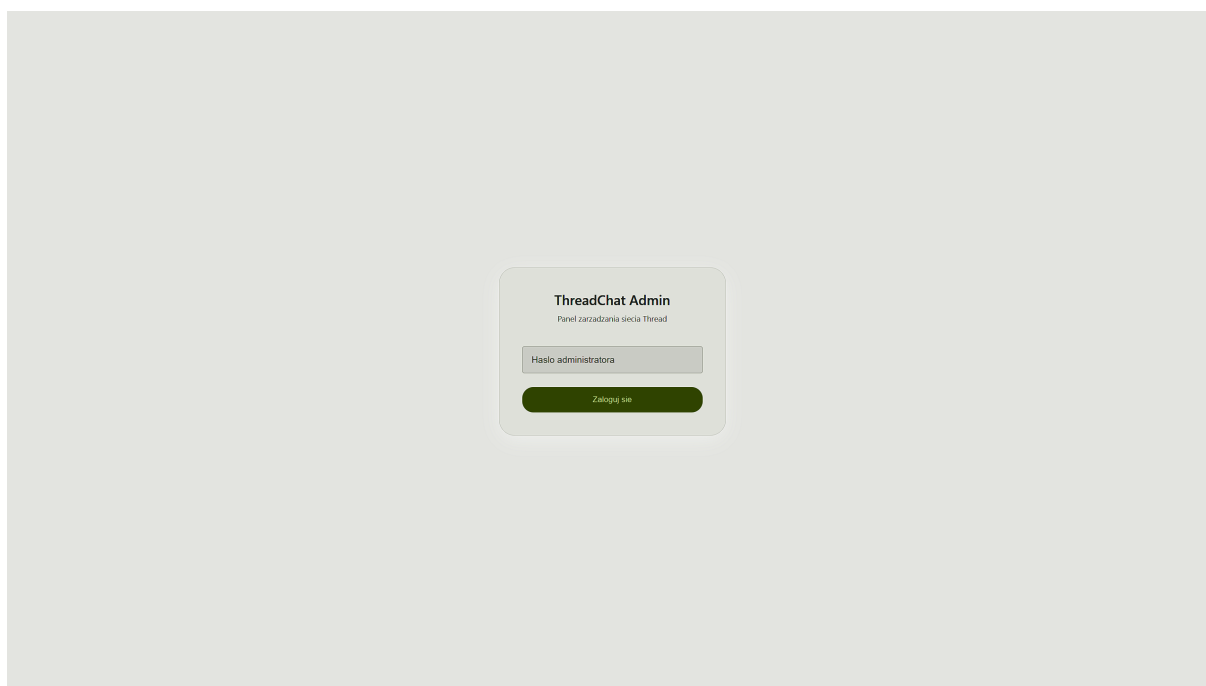
Dostęp do panelu chroniony jest hasłem administratora weryfikowanym przy logowaniu. Stan sesji przechowywany jest w atrybucie `admin_logged_in` obiektu `HttpSession` i sprawdzany przez pomocniczą metodę `isLoggedIn` na początku każdego punktu końcowego. Rysunek 6.1 przedstawia ekran logowania panelu.

Panel udostępnia następujące grupy funkcjonalności:

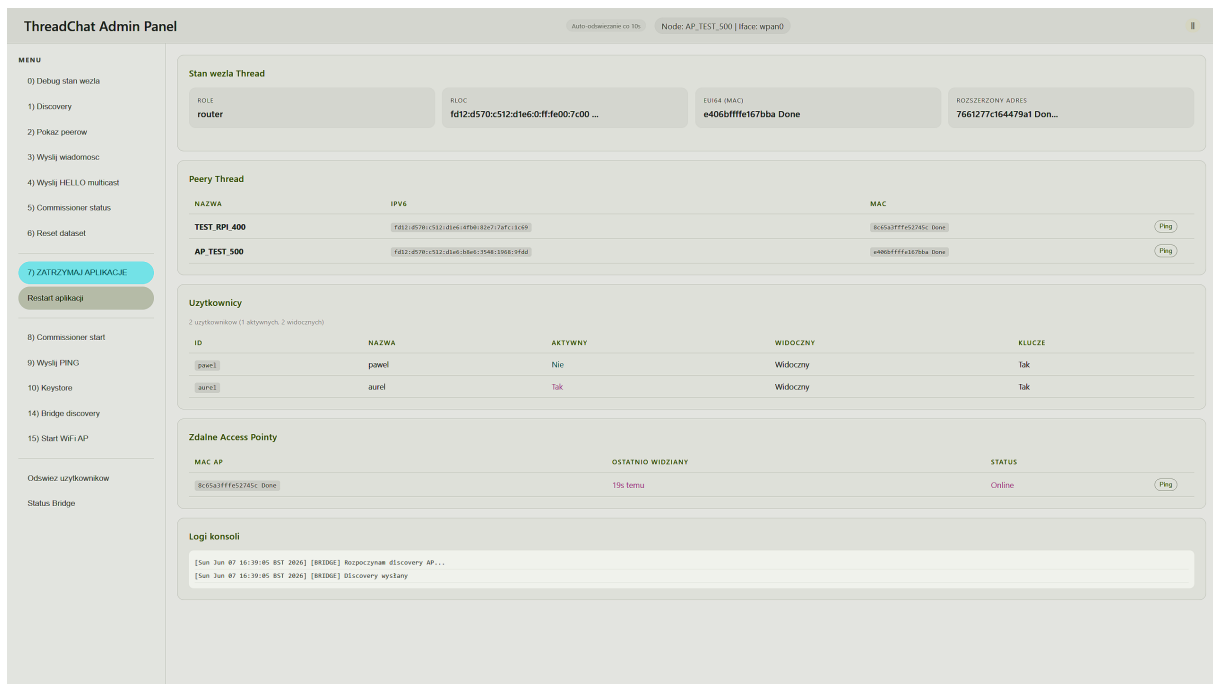
- **Stan węzła Thread** – endpoint `/api/state` zwraca aktualną rolę węzła w sieci (*leader*, *router*, *child*), adres RLOC, adres EUI64 oraz adres rozszerzony, odczytywane przez `0xCt1`;

- **Zarządzanie peerami** – endpoint `/api/peers` zwraca listę znanych węzłów sąsiednich z ich adresami IPv6 i MAC; endpoint `/api/discovery` inicjuje aktywne wykrywanie węzłów przez wysłanie ramki HELLO_REQ;
- **Zarządzanie siecią Thread** – endpoint `/api/commissioner-start` uruchamia komisjonera akceptującego nowe urządzenia; `/api/reset-dataset` czyści aktywny dataset sieci, co umożliwia ponowne dołączenie węzła do innej sieci bez ręcznej ingerencji;
- **Zarządzanie mostem** – endpoint `/api/bridge-aps` zwraca listę wykrytych zdalnych węzłów z czasem ostatniej aktywności; `/api/bridge-clients` raportuje liczbę i identyfikatory klientów lokalnych i zdalnych zarejestrowanych w ThreadFluxBridge;
- **Zarządzanie punktem dostępowym Wi-Fi** – endpoint `/api/wifi-start` wywołuje metodę `WifiAPManager.startAccessPoint()` opisaną w podrozdziale 6.6.1;
- **Zarządzanie cyklem życia aplikacji** – endpointy `/api/shutdown` i `/api/restart` umożliwiają zatrzymanie lub ponowne uruchomienie aplikacji bez dostępu przez SSH; shutdown wykonuje czysty shutdown Spring Boot, zatrzymuje NetService oraz demona Thread przed zakończeniem procesu;
- **Logi konsoli** – endpoint `/api/logs` zwraca ostatnie 100 wpisów logu gromadzonych przez metodę `addLog` i wyświetlanych w czasie rzeczywistym przez panel.

Rysunek 6.2 przedstawia widok główny panelu z widocznym stanem węzła Thread, listą peerów i logami konsoli.



Rysunek 6.1: Ekran logowania panelu administracyjnego. Źródło: Opracowanie własne



Rysunek 6.2: Widok główny panelu administracyjnego – stan węzła Thread, lista peerów i logi konsoli.
Źródło: Opracowanie własne

6.3 Implementacja backendu

Warstwa backendowa zorganizowana jest w trzech pakietach odpowiadających trzem poziomom abstrakcji. Pakiet `thread` zawiera kod niskopoziomowy: sterowanie demonem `ot-daemon` przez klasę `OtCtl`, transport UDP przez klasę `NetService` oraz klasę główną `ThreadChatIntegrated` uruchamiającą cały węzeł. Pakiet `flux` zawiera warstwę aplikacyjną Spring Boot: modele danych, serwisy, resolwery GraphQL oraz klasę `ThreadChatApplication`. Pakiet `bridge` zawiera klasę `ThreadFluxBridge` łączącą dwie pozostałe warstwy. Strukturę pakietów przedstawia rysunek 5.3.

Punktem wejścia warstwy Spring Boot jest klasa `ThreadChatApplication`, przedstawiona na listingu 6.8.

Listing 6.8: Klasa startowa Spring Boot – `ThreadChatApplication`

```
@SpringBootApplication
@EnableScheduling
@ComponentScan(basePackages = {
    "com.threadchat.flux.graphql",
    "com.threadchat.flux.service",
    "com.threadchat.flux.model",
    "com.threadchat.bridge",
})
public class ThreadChatApplication {
    public static void main(String[] args) {
        System.setProperty("java.net.preferIPv6Stack", "true");
        System.setProperty("java.net.preferIPv6Addresses", "true");
        SpringApplication.run(ThreadChatApplication.class, args);
    }
}
```

Adnotacja `@EnableScheduling` aktywuje mechanizm zadań cyklicznych, wykorzystywany przez `UserActivityManager` do okresowego sprawdzania aktywności użytkowników. Jawne wyliczenie pakietów w `@ComponentScan` jest konieczne, ponieważ pakiet `bridge` leży poza domyślnym zakresem skanowania Spring Boot – bez tego wpis bean `ThreadFluxBridge` nie zostałby wykryty przez kontener. Właściwości serwera zdefiniowane w pliku `application.properties` przedstawia listing 6.9.

Listing 6.9: Konfiguracja serwera – `application.properties`

```
spring.application.name=thread-chat
spring.graphql.websocket.path=/graphql
spring.graphql.path=/graphql
server.port=8080
```

Subskrypcje GraphQL transportowane są przez WebSocket pod tą samą ścieżką co zapytania HTTP, co upraszcza konfigurację klienta mobilnego – Apollo Kotlin łączy się z jednym adresem niezależnie od typu operacji.

6.3.1 Spring WebFlux

Podrozdział opisuje warstwę aplikacyjną Spring WebFlux: schemat GraphQL definiujący kontrakt API oraz resolwery obsługujące poszczególne operacje.

6.3.1.1 Schemat GraphQL

Kontrakt API zdefiniowany jest w pliku `schema.graphqls`. Listing 6.10 przedstawia pełną treść schematu.

Listing 6.10: Schemat GraphQL – `schema.graphqls`

```
type User {
  id: ID!
  displayName: String!
  publicKeys: PublicKeys!
  isActive: Boolean!
  isVisible: Boolean!
}

type PublicKeys {
  identityPublic: String!
  signedPreKeyPublic: String!
  oneTimePreKeys: [OneTimePreKey!]
  kyberPublicKey: String!
}

type OneTimePreKey {
  keyId: Int!
  publicKey: String!
}

type Message {
  id: ID!
  from: String!
  to: String!
  ciphertext: String!
  ratchetHeader: String!
  timestamp: String!
  iv: String!
  messageNumber: Int!
  ephemeralKey: String
  kyberCiphertext: String
}

type Query {
  getUser(id: ID!): User
  getUserKeys(userId: ID!): PublicKeys
  getMessages(to: ID!): [Message!]!
  getAllUsers: [User!]!
  getActiveUsers: [User!]!
  getInactiveUsers: [User!]!
}

type Mutation {
  publishKeys(
    userId: ID!
  )
}
```

```

        displayName: String!
        identityPublic: String!
        signedPreKeyPublic: String!
        oneTimePreKeys: [OneTimePreKeyInput!]
        isVisible: Boolean!
        kyberPublicKey: String!
    ): Boolean!

    sendMessage(
        from: String!
        to: String!
        ciphertext: String!
        ratchetHeader: String!
        iv: String!
        messageNumber: Int
        ephemeralKey: String
        kyberCiphertext: String
    ): Message!
}

type Subscription {
    onNewMessage(to: String!): Message!
}

input OneTimePreKeyInput {
    keyId: Int!
    publicKey: String!
}

```

Schemat definiuje cztery typy danych. Typ `User` reprezentuje uczestnika sieci – zawiera klucze publiczne X3DH (`PublicKeys`) oraz flagi aktywności i widoczności. Typ `Message` przenosi wyłącznie dane zaszyfrowane: pole `ciphertext` zawiera zaszyfrowaną treść, `ratchetHeader` nagłówek protokołu podwójnej zapadki [16] w formacie `base64:msgNum:prevLen`, a `iv` wektor inicjalizacji AES-GCM. Serwer nie ma dostępu do żadnego pola w postaci jawnej.

Operacje podzielone są na trzy grupy. Zapytania (*Query*) służą do pobierania danych o użytkownikach i ich kluczach. Mutacje (*Mutation*) obejmują publikację kluczy X3DH (`publishKeys`) oraz wysłanie wiadomości (`sendMessage`). Jedyna subskrypcja (`onNewMessage`) dostarcza wiadomości w czasie rzeczywistym przez `WebSocket`.

6.3.1.2 Resolwer wiadomości

Najbardziej złożonym fragmentem warstwy Flux jest resolwer `MessageResolver`. Listing 6.11 przedstawia mutację `sendMessage` realizującą routing wiadomości.

Listing 6.11: Routing wiadomości w mutacji `sendMessage`

```

@MutationMapping
public Message sendMessage(@Argument String from,
    @Argument String to, @Argument String ciphertext,
    @Argument String ratchetHeader, @Argument String iv,
    @Argument int messageNumber) {

    userManager.setUserActive(from);
    Message message = messageService.createMessage(
        from, to, ciphertext, ratchetHeader, iv, messageNumber);

    boolean isLocal = threadBridge.isLocalClient(to);
    boolean isRemote = threadBridge.isRemoteClient(to);

    if (isLocal) {
        if (messageService.isUserSubscribed(to)) {
            threadBridge.routeMessage(message); // natychmiast
        } else {
            messageCache.cacheMessage(message); // bufor
        }
    } else if (isRemote) {

```

```

        threadBridge.routeMessage(message);           // przez Thread
    } else {
        messageCache.cacheMessage(message);          // nieznany
    }
    return message;
}

```

Logika routingu opiera się na trzech przypadkach. Jeśli odbiorca jest klientem lokalnym z aktywną subskrypcją GraphQL, wiadomość trafia do niego natychmiast przez reaktywny strumień. Jeśli klient lokalny nie ma aktywnej subskrypcji – na przykład aplikacja jest w tle – wiadomość trafia do bufora `MessageCache` i zostanie dostarczona po ponownym nawiązaniu subskrypcji. Jeśli odbiorca jest klientem zdalnym, wiadomość przekazywana jest do `ThreadFluxBridge` i wysyłana przez sieć Thread do węzła docelowego.

Listing 6.12 przedstawia subskrypcję `onNewMessage`, będącą reaktywnym sercem systemu dostarczania wiadomości.

Listing 6.12: Subskrypcja `onNewMessage` – scalanie strumieni lokalnych i sieciowych

```

@SubscriptionMapping
public Publisher<Message> onNewMessage(@Argument String to) {
    userManager.setUserActive(to);
    threadBridge.registerLocalClient(to, displayName, isVisible);

    List<Message> pending = messageCache.getAndClearMessages(to);

    Flux<Message> newMessages =
        messageService.subscribeToMessages(to)
            .mergeWith(
                threadBridge.messageFlux()
                    .filter(msg -> msg.getTo().equals(to))
            )
            .doOnNext(msg ->
                userManager.refreshUserActivity(to))
            .doOnCancel(() -> {
                threadBridge.unregisterLocalClient(to);
                userManager.setUserInactive(to);
            });

    return pending.isEmpty()
        ? newMessages
        : Flux.fromIterable(pending).concatWith(newMessages);
}

```

Subskrypcja scala dwa niezależne strumienie reaktywne operatorem `mergeWith`: strumień wiadomości lokalnych z `MessageService` oraz strumień wiadomości przychodzących z sieci Thread przez `ThreadFluxBridge`. Dzięki temu klient mobilny otrzymuje wiadomości zarówno od użytkowników podłączonych do tego samego węzła, jak i od użytkowników zdalnych, przez jeden Websocket. Przed podłączeniem do strumienia nowych wiadomości dostarczane są wiadomości zbuforowane podczas nieobecności klienta – operator `concatWith` gwarantuje zachowanie kolejności. Anulowanie subskrypcji (rozłączenie klienta) automatycznie wyrejestrowuje go z `ThreadFluxBridge` i oznacza jako nieaktywnego.

6.3.1.3 Resolwer użytkowników

Listing 6.13 przedstawia mutację `publishKeys`, która inicjuje obecność użytkownika w całej sieci rozproszonej.

Listing 6.13: Mutacja `publishKeys` – rejestracja użytkownika w sieci

```

@MutationMapping
public Boolean publishKeys(@Argument String userId,
    @Argument String displayName,
    @Argument String identityPublic,
    @Argument String signedPreKeyPublic,
    @Argument List<OneTimePreKeyInput> oneTimePreKeys,

```

```

        @Argument Boolean isVisible,
        @Argument String kyberPublicKey) {

    PublicKeys keys = new PublicKeys();
    keys.setIdentityPublic(identityPublic);
    keys.setSignedPreKeyPublic(signedPreKeyPublic);
    keys.setOneTimePreKeys(oneTimePreKeys);
    keys.setKyberPublicKey(kyberPublicKey);

    boolean success = userService.publishKeys(
        userId, displayName, keys, isVisible, kyberPublicKey);

    if (success) {
        threadBridge.registerLocalClient(
            userId, displayName, isVisible);
        threadBridge.broadcastPublicKeys(userId, displayName,
            identityPublic, signedPreKeyPublic,
            oneTimePreKeys, kyberPublicKey);
    }
    return success;
}

```

Mutacja wykonuje trzy operacje sekwencyjnie. Najpierw klucze publiczne X3DH oraz klucz publiczny Kyber-768 zapisywane są lokalnie w `UserService`. Następnie użytkownik rejestrowany jest jako lokalny klient w `ThreadFluxBridge`, co pozwala routerowi wiadomości kierować do niego pakiety. Na końcu klucze rozsyłane są rozgłoszeniowo do wszystkich węzłów sieci Thread, dzięki czemu użytkownicy podłączeni do innych węzłów mogą zainicjować sesję kryptograficzną PQX3DH bez dodatkowych zapytań.

6.3.2 Thread

Podrozdział opisuje warstwę transportową pakietu `thread`: konfigurację węzła, binarny protokół ramkowania wiadomości UDP oraz mechanizm wykrywania i komunikacji z sąsiednimi węzłami.

6.3.2.1 Konfiguracja węzła

Parametry węzła przechowywane są w klasie `Config`, przedstawionej na listingu 6.14.

Listing 6.14: Parametry konfiguracyjne węzła – klasa `Config`

```

public class Config {
    public static String IFACE = "wpan0";
    public static String PSK = "J01NME";
    public static int PORT = 5678;
    public static String MCAST = "ff03::1";
    public static byte[] SECRET =
        "1234567890abcdef1234567890abcdef".getBytes();

    // przypisywane dynamicznie przez NativeBinaries
    public static String OTCTL;
    public static String OT_DAEMON_PATH;

    public static String RADIO_URL =
        "spinel+hdlc+uart:///dev/ttyUSB0" +
        "?uart-baudrate=460800,uart-reset";
    public static String THREAD_IFNAME = "wpan0";
}

```

Pole `IFACE` wskazuje wirtualny interfejs sieciowy Thread tworzony przez `ot-daemon`. Pole `MCAST` zawiera adres multicastowy `ff03::1` (*mesh-local all-nodes*), wykorzystywany do rozgłaszania ramek do wszystkich węzłów sieci. Pola `OTCTL` i `OT_DAEMON_PATH` pozostają puste do momentu uruchomienia – są przypisywane dynamicznie przez klasę `NativeBinaries` po wypakowaniu binarek z archiwum JAR, co opisano w podrozdziale 6.2.

6.3.2.2 Protokół ramkowania — MsgFrame

Komunikacja między węzłami opiera się na własnym binarnym protokole ramkowania zaimplementowanym w klasie `MsgFrame`. Każda ramka składa się z czterech bajtów magicznych TCH1 identyfikujących protokół, jednobajtowego pola typu, 16-bajtowego identyfikatora wiadomości (UUID), nazwy nadawcy poprzedzonej 2-bajtową długością oraz payloadu poprzedzonego 2-bajtową długością. Protokół definiuje cztery typy ramek: `HELLO_REQ` i `HELLO_RSP` służące do wykrywania węzłów i wymiany kluczy, `MSG` przenoszący wiadomość oraz `ACK` potwierdzający jej odbiór. Listing 6.15 przedstawia metodę budującą ramkę.

Listing 6.15: Budowanie ramki protokołu `MsgFrame`

```
public static byte[] buildFrame(byte type, byte[] msgId,
    String name, byte[] payload) {
    byte[] nameBytes = name == null
        ? new byte[0]
        : name.getBytes(StandardCharsets.UTF_8);
    byte[] pl = payload == null ? new byte[0] : payload;

    int len = 4 + 1 + 16 + 2 + nameBytes.length + 2 + pl.length;
    ByteBuffer bb = ByteBuffer.allocate(len);
    bb.put((byte)'T').put((byte)'C')
        .put((byte)'H').put((byte)'1'); // magic
    bb.put(type);
    bb.put(msgId);
    bb.putShort((short) nameBytes.length);
    bb.put(nameBytes);
    bb.putShort((short) pl.length);
    bb.put(pl);
    return bb.array();
}
```

Payload ramek `HELLO_REQ` i `HELLO_RSP` kodowany jest w formacie TLV (*Tag-Length-Value*) i przenosi trzy elementy: klucz publiczny X25519 (tag `0x01`, 32 bajty), adres mesh-local IPv6 nadawcy (tag `0x02`, 16 bajtów) oraz adres MAC EUI64 (tag `0x03`). Adres MAC EUI64 jest stabilnym identyfikatorem węzła niezależnym od aktualnego adresu RLOC, który może ulec zmianie przy rekonfiguracji topologii sieci.

6.3.2.3 Wykrywanie węzłów i nawiązywanie sesji

Klasa `NetService` zarządza gniazdami UDP i obsługuje cykl życia połączeń z sąsiednimi węzłami. Po uruchomieniu otwierane jest gniazdo unicastowe na adresie mesh-local węzła oraz gniazdo multicastowe nasłuchujące na adresie `ff03::1`. Wykrywanie sąsiadów inicjowane jest przez wysłanie ramki `HELLO_REQ` na adres multicastowy. Każdy węzeł, który ją odbierze, odpowiada ramką `HELLO_RSP` zawierającą własny klucz X25519.

Po odebraniu odpowiedzi `HELLO` węzeł wyznacza klucz sesji na poziomie transportowym. Listing 6.16 przedstawia tę procedurę.

Listing 6.16: Wyznaczanie klucza sesji transportowej po wymianie `HELLO`

```
private boolean helloMsgInitialSteps(
    InetAddress src, ParsedFrame pf) throws Exception {
    MsgFrame.HelloTLV tlv =
        MsgFrame.decodeHelloPayload(pf.payload());
    String macStr =
        new String(tlv.macEui64(), StandardCharsets.UTF_8);
    String ip = addrToMeshOrSrc(src, tlv.meshLocal());
    if (ip == null) return true;

    ipv6ToMac.put(ip, macStr);
    peers.put(macStr, new Peer(ip, pf.name(), macStr).touch());

    XECPublicKey peerPub =
        KeyUtil.decodeX25519Public(tlv.pub());
    byte[] shared =
```

```

        KeyUtil.x25519Shared(myIdentity.getPrivate(), peerPub);
byte[] sessionKey =
    CryptoUtils.hkdf(shared, "CHAT_SESSION");
sessions.put(macStr, new Session(peerPub, sessionKey));
return false;
}

```

Metoda wyciąga klucz publiczny X25519 peera z payloadu TLV, oblicza wspólny sekret metodą Diffie-Hellman na krzywej X25519, a następnie wyprowadza 32-bajtowy klucz sesji funkcją HKDF z etykietą CHAT_SESSION. Sesja indeksowana jest po adresie MAC EUI64, a nie po adresie IPv6, co zapewnia ciągłość sesji przy zmianie adresu RLOC węzła.

Opisany mechanizm stanowi warstwę szyfrowania na poziomie transportowym między węzłami Raspberry Pi. Należy zaznaczyć, że właściwe szyfrowanie wiadomości użytkowników realizowane jest niezależnie przez protokół PQX3DH i protokół podwójnej zapadki po stronie aplikacji mobilnej, opisany w podrozdziale 6.4.2. Szyfrowanie transportowe między węzłami było elementem wcześniejszego etapu projektowania i nie jest wymuszone przez architekturę - wiadomości aplikacyjne docierają do węzłów już zaszyfrowane przez klienta mobilnego.

6.3.2.4 Synchronizacja czasu sieci rozproszonej

Węzły sieci nie mają gwarancji zsynchronizowanego zegara systemowego, co mogłoby prowadzić do niespójności znaczników czasu wiadomości. Klasa MeshTimeSync implementuje prosty mechanizm synchronizacji czasu przez sieć rozproszoną: węzeł zbiera znaczniki czasu od sąsiadów i wyznacza czas docelowy jako średnią ważoną, preferując węzły o mniejszej liczbie hopów od korzenia topologii. Listing 6.17 przedstawia metodę obliczającą tę średnią.

Listing 6.17: Wyznaczanie czasu docelowego jako średniej ważonej odpowiedzi sąsiadów

```

private static long calculateAverageTime(
    List<TimeResponse> responses) {
    List<TimeResponse> sorted = new ArrayList<>(responses);
    sorted.sort((a, b) ->
        Integer.compare(a.hopCount, b.hopCount));

    int minHop = sorted.get(0).hopCount;
    long weightedSum = 0;
    int weightedCount = 0;
    for (TimeResponse r : sorted) {
        int weight = (r.hopCount == minHop) ? 3 : 1;
        weightedSum += r.timestamp * weight;
        weightedCount += weight;
    }
    boolean varied = sorted.get(sorted.size()-1).hopCount
        != minHop;
    if (varied) return weightedSum / weightedCount;

    long sum = 0;
    for (TimeResponse r : responses) sum += r.timestamp;
    return sum / responses.size();
}

```

Węzły bezpośrednio sąsiadujące z węzłem o najniższej liczbie przeskoków otrzymują wagę trzykrotnie wyższą niż pozostałe. Jeśli wszystkie węzły mają tę samą liczbę przeskoków – co jest typowe w płaskiej topologii testowej – stosowana jest prosta średnia arytmetyczna. Wyznaczony czas ustawiany jest poleceniem systemowym `sudo date`.

6.3.3 ThreadFluxBridge

Klasa ThreadFluxBridge stanowi most między warstwą transportową Thread a reaktywnym potokiem Spring WebFlux. Jest beanem Spring (@Service), lecz operuje bezpośrednio na gniazdach UDP sieci rozproszonej. Jej zadaniem jest tłumaczenie pakietów UDP odbieranych z sieci Thread na obiekty dome-

nowe Javy emitowane do strumienia reaktywnego, z którego korzystają resolwery GraphQL obsługujące połączenia aplikacji mobilnych.

6.3.3.1 Inicjalizacja i gniazda

Klasa utrzymuje dwa osobne gniazda UDP. `MulticastSocket` bindowany do `[::]:5555` służy wyłącznie do odbierania ramek – nasłuchuje na wszystkich interfejsach. Osobny `DatagramSocket` bindowany do adresu MLEID węzła na porcie 5556 służy wyłącznie do wysyłania, co zapewnia prawidłowy adres źródłowy pakietów wychodzących.

Inicjalizacja uruchamiana jest przez metodę oznaczoną adnotacją `@PostConstruct`, jednak nie odbywa się synchronicznie – metoda uruchamia osobny wątek z celowym opóźnieniem 2 sekund przed właściwą inicjalizacją. Wynika to ze świadomego rozdzielenia cykli życia obu komponentów: `NetService` uruchamiany jest przez klasę `ThreadChatIntegrated` w osobnym wątku, niezależnie od kontenera Spring. Opóźnienie daje `NetService` czas na pełne uruchomienie przed próbą pobrania jego instancji przez `ThreadFluxBridge`.

6.3.3.2 Protokół komunikacji między węzłami – BridgeFrame

Węzły komunikują się za pomocą serializowanych ramek JSON przez bibliotekę Jackson i przesyłanych przez UDP. Klasa `BridgeFrame` definiuje strukturę ramki oraz dziewięć typów wiadomości. Jest to wyższy poziom abstrakcji niż binarny protokół `MsgFrame` używany w warstwie transportowej - zamiast własnego formatu binarnego stosowany jest JSON, co upraszcza rozszerzanie protokołu o nowe pola. Typy ramek wraz z ich przeznaczeniem zestawiono w tabeli 6.4.

Tabela 6.4: Typy ramek protokołu `BridgeFrame` i ich przeznaczenie

Typ	Przeznaczenie
MESSAGE	Przesłanie zaszyfrowanej wiadomości użytkownika
CLIENT_HELLO	Rejestracja klienta mobilnego na zdalnym węźle
CLIENT_BYE	Wyrejestrowanie klienta po rozłączeniu
DISCOVERY_REQUEST	Żądanie wykrycia zdalnych węzłów w sieci
DISCOVERY_RESPONSE	Odpowiedź na żądanie wykrycia
AP_ANNOUNCEMENT	Ogłoszenie obecności węzła w sieci rozproszonej
KEYS_PUBLISHED	Rozgłoszenie kluczy publicznych X3DH użytkownika
CLIENT_ACTIVE_STATUS	Aktualizacja statusu aktywności użytkownika
TIME_SYNC	Synchronizacja czasu systemowego między węzłami

Listing 6.18 przedstawia definicję klasy `BridgeFrame`.

Listing 6.18: Struktura ramki protokołu `BridgeFrame`

```
public static class BridgeFrame {
    public enum Type {
        MESSAGE, CLIENT_HELLO, CLIENT_BYE,
        DISCOVERY_REQUEST, DISCOVERY_RESPONSE,
        AP_ANNOUNCEMENT, KEYS_PUBLISHED,
        CLIENT_ACTIVE_STATUS, TIME_SYNC
    }
    public Type type;
    public Message message;
    public String userId;
    public String displayName;
    public String senderMac; // MAC EUI64 nadawcy
    public String senderRloc; // RLOC IPv6 nadawcy
    public Boolean isActive;
    public Boolean isVisible;
    public String kyberPublicKey;
    // klucze publiczne X3DH
    public String identityPublic;
    public String signedPreKeyPublic;
    public List<OneTimePreKeyInput> oneTimePreKeys;
```

```

// synchronizacja czasu
public Long    timestamp;
public Integer hopCount;
}

```

Każda ramka zawiera pola `senderMac` i `senderRloc` identyfikujące nadawcę. Adres MAC EUI64 jest stabilnym identyfikatorem sprzętowym niezależnym od topologii sieci, natomiast RLOC (*Routing Locator*) jest adresem routingowym, który może ulec zmianie przy rekonfiguracji sieci Thread. Rozróżnienie to jest kluczowe dla poprawnego adresowania – węzły identyfikowane są po MAC, lecz pakiety wysyłane są na aktualny RLOC.

6.3.3.3 Zarządzanie klientami

Most utrzymuje w pamięci operacyjnej dwie rozłączne kolekcje klientów. Mapa `localClients` zawiera użytkowników podłączonych bezpośrednio do bieżącego węzła przez `WebSocket`. Mapa `remoteClients` zawiera użytkowników zarejestrowanych na węzłach zdalnych, których obecność znana jest dzięki ramkom `CLIENT_HELLO` rozsyłanym przez inne węzły. Obie kolekcje są rozłączne – rejestracja użytkownika jako lokalnego usuwa go ze zdalnych i odwrotnie. Listing 6.19 przedstawia metodę rejestrującą lokalnego klienta.

Listing 6.19: Rejestracja lokalnego klienta w `ThreadFluxBridge`

```

public void registerLocalClient(String userId,
    String displayName, boolean isVisible) {
    LocalClient existing = localClients.get(userId);
    if (existing != null) {
        existing.isVisible = isVisible;
        if (initialized.get()) {
            broadcastClientPresence(
                userId, displayName, true, isVisible);
        }
        return;
    }
    LocalClient client = new LocalClient(userId, displayName,
        System.currentTimeMillis(), isVisible);
    localClients.put(userId, client);
    remoteClients.remove(userId);
    broadcastClientPresence(userId, displayName, true, isVisible);
}

```

Po rejestracji nowego klienta jego obecność rozsyłana jest rozgłoszeniowo do wszystkich znanych węzłów przez ramkę `CLIENT_HELLO`, dzięki czemu każdy węzeł sieci posiada aktualną informację o dostępności użytkowników.

6.3.3.4 Routing wiadomości

Centralna metoda `routeMessage` podejmuje decyzję o sposobie dostarczenia wiadomości na podstawie lokalizacji odbiorcy, przedstawioną na listingu 6.20.

Listing 6.20: Routing wiadomości w `ThreadFluxBridge`

```

public boolean routeMessage(Message message) {
    String recipientId = message.getTo();

    if (localClients.containsKey(recipientId)) {
        messageSink.tryEmitNext(message);    // lokalny
        return true;
    }

    RemoteClient rc = remoteClients.get(recipientId);
    if (rc != null && initialized.get()) {
        RemoteAP ap = remoteAccessPoints.get(rc.apMac);
        if (ap != null) {
            return sendToRemoteAP(ap.getRLOC(), message);
        }
    }
}

```

```

    }

    if (messageCache != null) {
        messageCache.cacheMessage(message); // nieznany
        return true;
    }
    messageSink.tryEmitNext(message);      // fallback
    return true;
}

```

Wiadomość przeznaczona dla klienta lokalnego emitowana jest bezpośrednio do reaktywnego `Sinks.Many`, z którego korzysta subskrypcja `onNewMessage`. Wiadomość przeznaczona dla klienta zdalnego serializowana jest do JSON i wysyłana unicastem na adres RLOC węzła docelowego przez `sendSocket`. Dla odbiorcy o nieznanej lokalizacji wiadomość trafia do bufora `MessageCache`.

6.3.3.5 Adresowanie zdalnych węzłów

Zdalne węzły reprezentowane są przez klasę pomocniczą `RemoteAP` indeksowaną po adresie MAC EUI64. Każdy wpis przechowuje aktualny adres RLOC używany do wysyłania pakietów. Ponieważ adresy RLOC mogą ulec zmianie przy rekonfiguracji topologii sieci Thread, zadanie cykliczne uruchamiane co 30 sekund odświeża je na podstawie tablicy routerów odczytywanej przez `OtCt1`. Identyfikacja węzłów opiera się na stabilnym adresie MAC EUI64, natomiast aktualny RLOC służy wyłącznie jako adres docelowy pakietów UDP.

6.3.3.6 Synchronizacja czasu

Węzły sieci rozproszonej nie mają gwarancji zsynchronizowanego zegara systemowego. Przy wykryciu nowego węzła `ThreadFluxBridge` wysyła do niego ramkę `TIME_SYNC` zawierającą bieżący znacznik czasu oraz wartość `hopCount` określającą odległość węzła od korzenia topologii. Odbiorca przekazuje otrzymane dane do klasy `MeshTimeSync`, która synchronizuje zegar systemowy poleceniem `sudo date`. Mechanizm `hopCount` pozwala preferować węzły bliżej korzenia jako bardziej wiarygodne źródło czasu – opisano go szczegółowo w podrozdziale 6.3.2.

6.4 Implementacja aplikacji mobilnej

Aplikacja mobilna napisana jest w języku Kotlin na platformę Android z interfejsem opartym na Jetpack Compose. Poniższe podrozdziały opisują warstwę lokalnego przechowywania danych, która jest realizowana przy pomocy biblioteki Room [20], moduł kryptograficzny realizujący protokół PQX3DH i protokół podwójnej zapadki oraz interfejs użytkownika.

6.4.1 Room

Dane aplikacji przechowywane są lokalnie z użyciem biblioteki Room [20] – warstwy abstrakcji nad SQLite. Zastosowano dwie niezależne bazy danych różniące się zakresem danych i cyklem życia.

6.4.1.1 Encje danych

Encja `UserEntity` przechowuje informacje o znanych uczestnikach sieci: identyfikator, nazwę wyświetlaną i znacznik czasu ostatniej interakcji. Dane te są trwałe – przeżywają restart aplikacji i są dostępne bez połączenia z węzłem. Listing 6.21 przedstawia definicję tej encji.

Listing 6.21: Encja `UserEntity` – globalna baza użytkowników

```

@Entity(tableName = "users")
data class UserEntity(
    @PrimaryKey val userId: String,
    val displayName: String,
    val lastInteraction: Long = System.currentTimeMillis()
)

```

Encja `MessageEntity` przechowuje wiadomości z pełną strukturą kryptograficzną wymaganą przez protokół podwójnej zapadki. Listing 6.22 przedstawia jej definicję.

Listing 6.22: Encja `MessageEntity` – struktura kryptograficzna wiadomości

```
@Entity(tableName = "messages")
data class MessageEntity(
    @PrimaryKey val id: String,
    val from: String,
    val to: String,
    val ciphertext: String,
    val timestamp: String,
    val ratchetHeader: String? = null, // base64Key:msgNum:prevChainLen
    val iv: String? = null,
    // klucz wiadomosci zaszyfrowany przez Android Keystore
    // (tylko dla wiadomosci wychodzących)
    val encryptedMessageKey: String? = null,
    val encryptedMessageKeyIv: String? = null,
    val isLocallyEncrypted: Boolean = false,
    val messageNumber: Int = 0
)
```

Pole `ratchetHeader` zawiera nagłówek protokołu podwójnej zapadki w formacie `base64Key:msgNum:prevChainLen`, niezbędny do odtworzenia stanu protokołu przy odszyfrowaniu. Pola `encryptedMessageKey` i `encryptedMessageKeyIv` przechowują klucz wiadomości zaszyfrowany kluczem głównym z Magazynu kluczy Android – wyłącznie dla wiadomości wysłanych przez bieżącego użytkownika. Dla wiadomości przychodzących pola te są puste, a klucz odtwarzany jest ze stanu podwójnej zapadki.

6.4.1.2 Izolacja rozmów

Każda para użytkowników posiada odrębną bazę danych SQLite o nazwie `chat_<id1>_<id2>.db`. Identyfikatory sortowane są leksykograficznie, co gwarantuje tę samą nazwę niezależnie od tego, która strona inicjuje rozmowę. Tworzeniem i zarządzaniem instancjami zajmuje się klasa `DatabaseProvider`, przedstawiona na listingu 6.23.

Listing 6.23: Tworzenie izolowanej bazy danych dla pary rozmówców – klasa `DatabaseProvider`

```
fun getDatabase(context: Context,
               userId1: String,
               userId2: String): AppDatabase {
    val dbName = getDatabaseName(userId1, userId2)
    return synchronized(this) {
        instances[dbName] ?: createDatabase(context, dbName)
        .also { instances[dbName] = it }
    }
}

private fun getDatabaseName(userId1: String,
                           userId2: String): String {
    val sortedIds = listOf(userId1, userId2).sorted()
    return "chat_${sortedIds[0]}_${sortedIds[1]}.db"
}
```

Sortowanie identyfikatorów przed złączeniem zapewnia deterministyczną nazwę bazy – para (Alice, Bob) i (Bob, Alice) zawsze odwołuje się do tego samego pliku. Usunięcie rozmowy sprowadza się do usunięcia jednego pliku bez ryzyka naruszenia pozostałych konwersacji. Wykrywanie istniejących rozmów odbywa się przez listowanie plików w katalogu baz danych i filtrowanie wzorcem `chat_*.db`, bez potrzeby centralnej tabeli katalogu.

Globalna baza `users_global.db` przechowuje encje `UserEntity` współdzielone między rozmowami. Rozdzielenie to upraszcza zarządzanie cyklem życia danych: usunięcie rozmowy nie usuwa informacji o użytkowniku z listy kontaktów.

6.4.1.3 Warstwa dostępu do danych

Dostęp do baz danych realizowany jest przez interfejsy DAO. `MessageDao` udostępnia operacje na wiadomościach, a `UserDao` na użytkownikach. Zapytania zwracają obiekty reaktywne `Flow` z biblioteki Kotlin Coroutines, dzięki czemu interfejs automatycznie reaguje na zmiany danych bez jawnego odpytania bazy. Listing 6.24 przedstawia wybrane metody `MessageDao`.

Listing 6.24: Wybrane metody interfejsu `MessageDao`

```
@Dao
interface MessageDao {
    @Query("SELECT * FROM messages ORDER BY timestamp ASC")
    fun getAllMessages(): Flow<List<MessageEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertMessage(message: MessageEntity)

    @Query("UPDATE messages SET encryptedMessageKey = :key,
            encryptedMessageKeyIv = :iv WHERE id = :id")
    suspend fun updateEncryptedKey(
        id: String, key: ByteArray?, iv: ByteArray?)
}
```

Metoda `getAllMessages` zwraca `Flow` emitujący nową listę przy każdej zmianie w tabeli. Metoda `updateEncryptedKey` umożliwia późniejsze dopisanie zaszyfrowanego klucza wiadomości po uwierzytelnieniu biometrycznym, bez modyfikacji pozostałych pól rekordu.

6.4.2 Protokół PQX3DH

Moduł kryptograficzny aplikacji mobilnej realizuje dwa niezależne protokoły. Protokół PQX3DH odpowiada za jednorazowe uzgodnienie klucza sesji na początku rozmowy. Protokół podwójnej zapadki zapewnia szyfrowanie każdej wiadomości osobnym kluczem z właściwością doskonałej tajemnicy i bezpieczeństwa po naruszeniu. Do operacji niskopoziomowych wykorzystywana jest biblioteka Bouncy Castle [32] w wersji 1.82.

6.4.2.1 Protokół PQX3DH

Protokół PQX3DH stanowi rozszerzenie standardowego protokołu X3DH [15] o mechanizm enkapsulacji klucza oparty na algorytmie Kyber-768 (*Key Encapsulation Mechanism*, KEM). Kyber-768 należy do rodziny algorytmów CRYSTALS wybranych przez NIST w procesie standaryzacji kryptografii postkwantowej i jest odporny na ataki z wykorzystaniem komputerów kwantowych.

Uzgodnienie klucza sesji przebiega w dwóch fazach. Po stronie inicjatora listing 6.25 przedstawia metodę `initiatorHandshake`.

Listing 6.25: Uzgodnienie PQX3DH po stronie inicjatora

```
fun initiatorHandshake(
    myIdentityKey: KeyPair,
    myOneTimePreKey: KeyPair?,
    theirIdentityKey: PublicKey,
    theirSignedPreKey: PublicKey,
    theirOneTimePreKey: PublicKey?,
    theirKyberPublicKey: KyberPublicKeyParameters
): PQX3DHInitiatorResult {
    val ephemeralKeyPair = CryptoUtils.generateX25519KeyPair()

    val dh1 = CryptoUtils.performX25519KeyExchange(
        myIdentityKey.private, theirSignedPreKey)
    val dh2 = CryptoUtils.performX25519KeyExchange(
        ephemeralKeyPair.private, theirIdentityKey)
    val dh3 = CryptoUtils.performX25519KeyExchange(
        ephemeralKeyPair.private, theirSignedPreKey)
    val dh4: ByteArray = if (myOneTimePreKey != null
        && theirOneTimePreKey != null)
```

```

        CryptoUtils.performX25519KeyExchange(
            ephemeralKeyPair.private, theirOneTimePreKey)
    else ByteArray(32)

    val kyberResult = CryptoUtils.kyberEncapsulate(
        theirKyberPublicKey)

    val combinedSecret =
        dh1 + dh2 + dh3 + dh4 + kyberResult.sharedSecret
    val sessionKey = CryptoUtils.kdf(
        combinedSecret, "PQX3DH_SESSION_KEY_v2", 32)

    return PQX3DHInitiatorResult(
        sessionKey = sessionKey,
        ephemeralKey = ephemeralKeyPair.public.encoded,
        kyberCiphertext = kyberResult.ciphertext
    )
}

```

Inicjator generuje efemeryczny klucz X25519 unikalny dla każdej sesji, oblicza cztery wymiany Diffie-Hellman na krzywej X25519 zgodnie ze specyfikacją Signal X3DH, a następnie enkapsuluje dodatkowy sekret kluczem publicznym Kyber-768 odbiorcy. Wszystkie pięć sekretów (DH1–DH4 i sekret Kyber) konkatelowane są w stałej kolejności i przepuszczane przez funkcję KDF z etykietą PQX3DH_SESSION_KEY_v2. Inicjator wysyła odbiorcy klucz efemeryczny EK_A.pub oraz kyberCiphertext dołączone do pierwszej wiadomości.

Po stronie odbiorcy metoda responderHandshake wykonuje lustrzane operacje: oblicza te same cztery wymiany DH, używając otrzymanego EK_A.pub, a następnie dekapsuluje kyberCiphertext własnym kluczem prywatnym Kyber-768. Identyczna kolejność konkatencji sekretów i ten sam label KDF gwarantują uzyskanie identycznego klucza sesji po obu stronach.

Implementacja Kyber-768 opiera się na klasach KyberKEMGenerator i KyberKEMExtractor z biblioteki Bouncy Castle [32]. Listing 6.26 przedstawia obie operacje KEM.

Listing 6.26: Enkapsulacja i dekapsulacja Kyber-768 w klasie CryptoUtils

```

fun kyberEncapsulate(
    recipientPublicKey: KyberPublicKeyParameters
): KyberEncapsulation {
    val kemGen = KyberKEMGenerator(secureRandom)
    val encapsulated =
        kemGen.generateEncapsulated(recipientPublicKey)
    return KyberEncapsulation(
        ciphertext = encapsulated.encapsulation,
        sharedSecret = encapsulated.secret
    )
}

fun kyberDecapsulate(
    privateKey: KyberPrivateKeyParameters,
    ciphertext: ByteArray
): ByteArray =
    KyberKEMExtractor(privateKey).extractSecret(ciphertext)

```

6.4.2.2 Protokół podwójnej zapadki

Po uzgodnieniu klucza sesji dalsza komunikacja prowadzona jest protokołem podwójnej zapadki [16]. Każda wiadomość szyfrowana jest algorytmem AES-256-GCM z unikalnym kluczem wyprowadzanym z bieżącego stanu łańcucha kluczy. Stan protokołu przechowywany jest w strukturze RatchetState zawierającej klucz korzenia (rootKey), klucze łańcuchów wysyłania i odbierania oraz bieżącą parę kluczy DH.

Protokół realizuje dwa niezależne mechanizmy rotacji kluczy. Symetryczny mechanizm *KDF chain* wyprowadza z każdego klucza łańcucha nowy klucz łańcucha i klucz wiadomości przez HMAC-SHA256, co zapewnia utajnienie z wyprzedzeniem w obrębie jednego łańcucha. Asymetryczny mechanizm zapad-

kowy Diffie-Hellmana wymienia pary kluczy przy każdej zmianie kierunku komunikacji, co zapewnia utajnienie z wyprzedzeniem. Listing 6.27 przedstawia implementację funkcji wyprowadzania klucza łańcucha.

Listing 6.27: Funkcja wyprowadzania klucza wiadomości z klucza łańcucha (kdfCK)

```
private fun kdfCK(chainKey: ByteArray?): Pair<ByteArray, ByteArray> {
    val hmac = Mac.getInstance("HmacSHA256")
    hmac.init(SecretKeySpec(chainKey, "HmacSHA256"))
    val messageKey = hmac.doFinal(byteArrayOf(0x01))

    hmac.init(SecretKeySpec(chainKey, "HmacSHA256"))
    val newChainKey = hmac.doFinal(byteArrayOf(0x02))

    return Pair(newChainKey, messageKey)
}
```

Stałe 0x01 i 0x02 rozdzielają wyprowadzenie klucza wiadomości od wyprowadzenia nowego klucza łańcucha, zgodnie ze specyfikacją protokołu. Klucz wiadomości jest niszczone po użyciu – nie jest przechowywany w stanie RatchetState.

Nagłówek każdej wiadomości zawiera klucz publiczny DH nadawcy, numer wiadomości w łańcuchu i długość poprzedniego łańcucha, serializowany w formacie `base64Key:msgNum:prevChainLen`. Umożliwia to odbiorcy wykrycie zmiany klucza i wykonanie kroku DH ratchet, a także obsługę wiadomości dostarczonych poza kolejnością przez buforowanie kluczy pominiętych wiadomości w mapie `skippedMessageKeys`.

6.4.2.3 Ochrona kluczy – Magazyn kluczy Android

Klucze kryptograficzne wiadomości wysłanych przez bieżącego użytkownika są dodatkowo szyfrowane kluczem głównym przechowywanym w Magazynie kluczy. Klucz główny o aliasie `MessageKeyMaster` generowany jest algorytmem AES-256-GCM z wymaganym uwierzytelnieniem użytkownika. Listing 6.28 przedstawia jego konfigurację.

Listing 6.28: Konfiguracja klucza głównego w Android Keystore

```
val spec = KeyGenParameterSpec.Builder(
    MASTER_KEY_ALIAS,
    KeyProperties.PURPOSE_ENCRYPT or
    KeyProperties.PURPOSE_DECRYPT
)
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(
        KeyProperties.ENCRYPTION_PADDING_NONE
    )
    .setUserAuthenticationRequired(true)
    .setUserAuthenticationParameters(
        600,
        KeyProperties.AUTH_BIOMETRIC_STRONG or
        KeyProperties.AUTH_DEVICE_CREDENTIAL
    )
    .build()
```

Parametr `setUserAuthenticationRequired(true)` wymusza potwierdzenie tożsamości przed każdą operacją kryptograficzną. Okno ważności uwierzytelnienia wynosi 600 sekund – po jego upływie każda próba odszyfrowania wiadomości inicjuje wyświetlenie monitu biometrycznego. Uwierzytelnieniem biometrycznym zarządza klasa `BiometricAuthManager`, opakowująca asynchroniczne API `BiometricPrompt` w korutynę Kotlin. Obsługiwane są odcisk palca, rozpoznanie twarzy oraz PIN urządzenia jako mechanizm zastępczy.

6.4.3 UI/UX

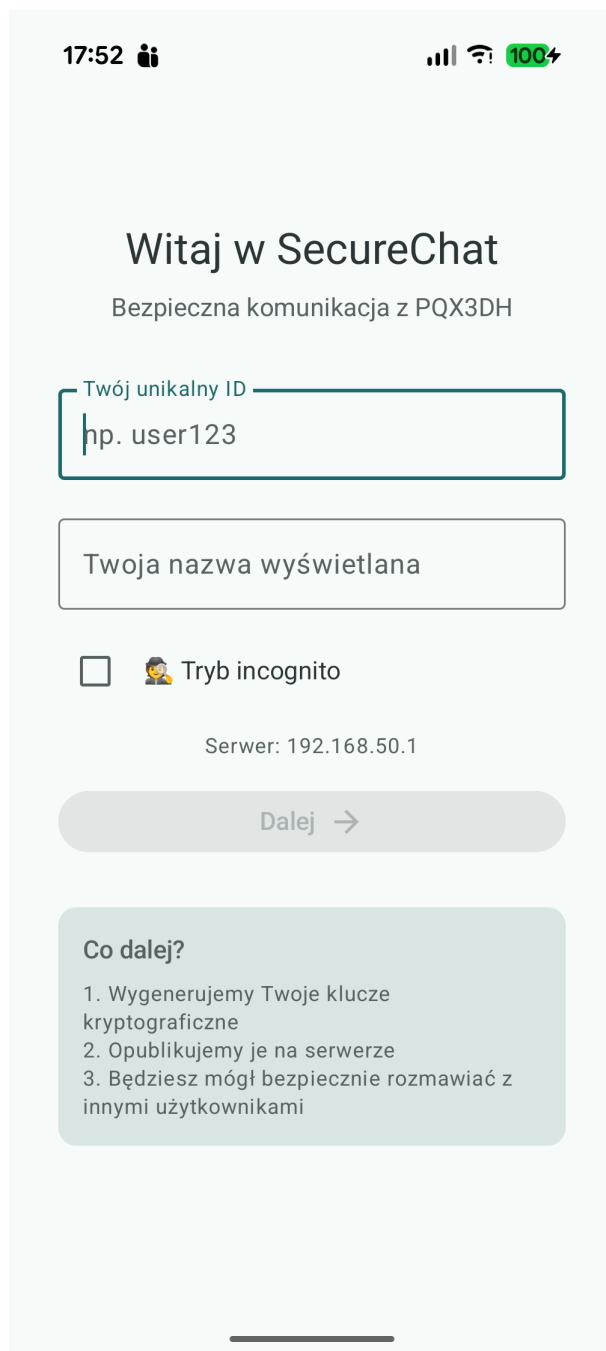
Interfejs użytkownika zbudowany jest w całości w Jetpack Compose z nawigacją opartą na bibliotece `androidx.navigation.compose`. Punkt wejścia aplikacji stanowi klasa `AppEntryPoint`, która na

podstawie stanu sesji i ewentualnej intencji z powiadomienia systemowego wyznacza ekran startowy. Aplikacja obsługuje trzy ścieżki uruchomienia: pierwsze uruchomienie bez zapisanej sesji kieruje do ekranu rejestracji, kolejne uruchomienie z zapisaną sesją pomija rejestrację i przechodzi bezpośrednio do listy kontaktów, a kliknięcie powiadomienia o nowej wiadomości otwiera bezpośrednio okno rozmowy z nadawcą.

6.4.3.1 Ekran aplikacji

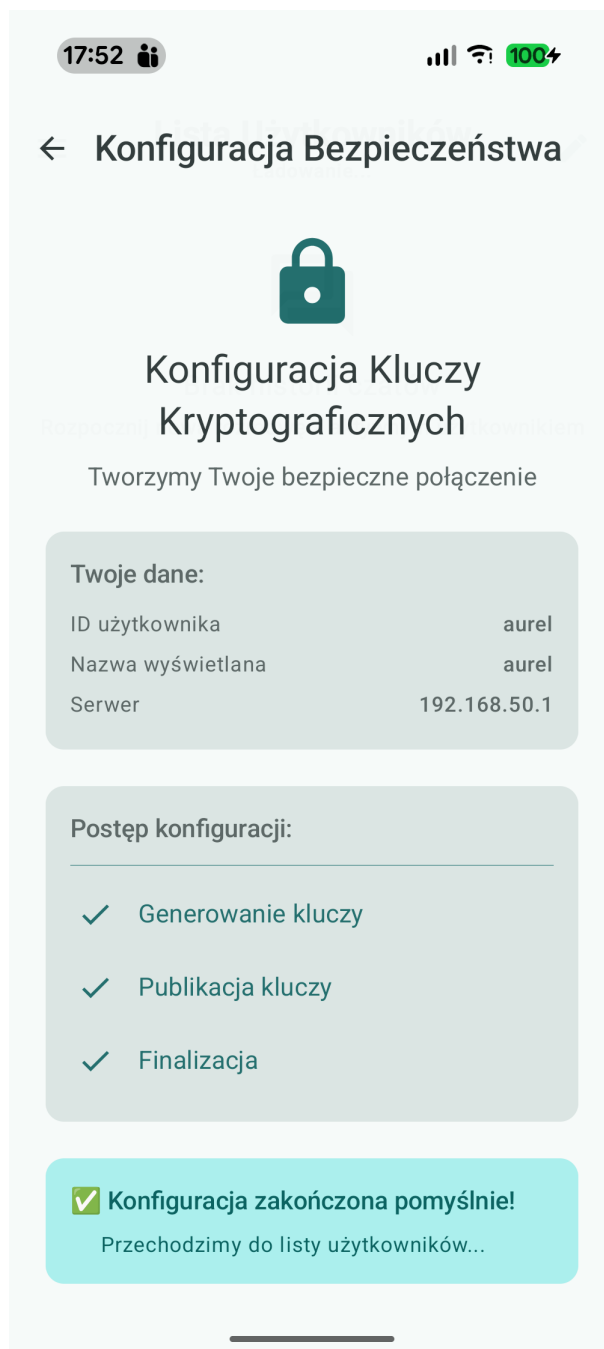
Aplikacja składa się z pięciu głównych ekranów.

Ekran **rejestracji** (`SetNameScreen`) umożliwia wprowadzenie unikalnego identyfikatora użytkownika, nazwy wyświetlanej oraz wybór trybu incognito. Tryb incognito powoduje, że użytkownik nie jest widoczny na listach kontaktów innych uczestników, lecz nadal może uczestniczyć w komunikacji. Rysunek 6.3 przedstawia wygląd tego ekranu.



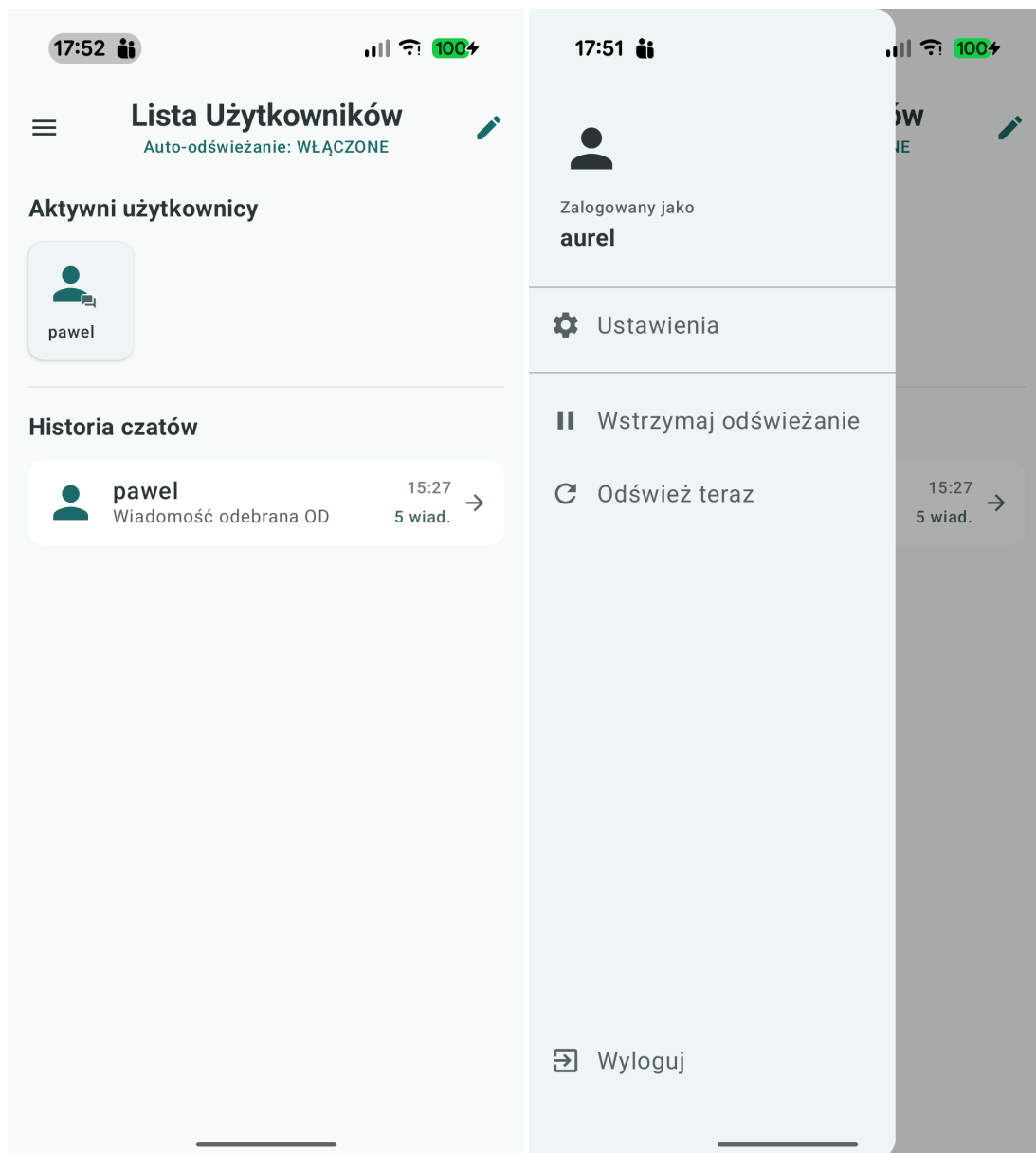
Rysunek 6.3: Ekran rejestracji użytkownika – SetNameScreen. Źródło: Opracowanie własne

Ekran **konfiguracji kluczy** (KeySetupScreen) uruchamiany jednorazowo po rejestracji. Generuje parę kluczy X25519 (klucz tożsamości, klucz podpisany, klucze jednorazowe) oraz parę kluczy Kyber-768, a następnie publikuje klucze publiczne na węźle przez mutację GraphQL `publishKeys`. Rysunek 6.4 przedstawia wygląd tego ekranu.



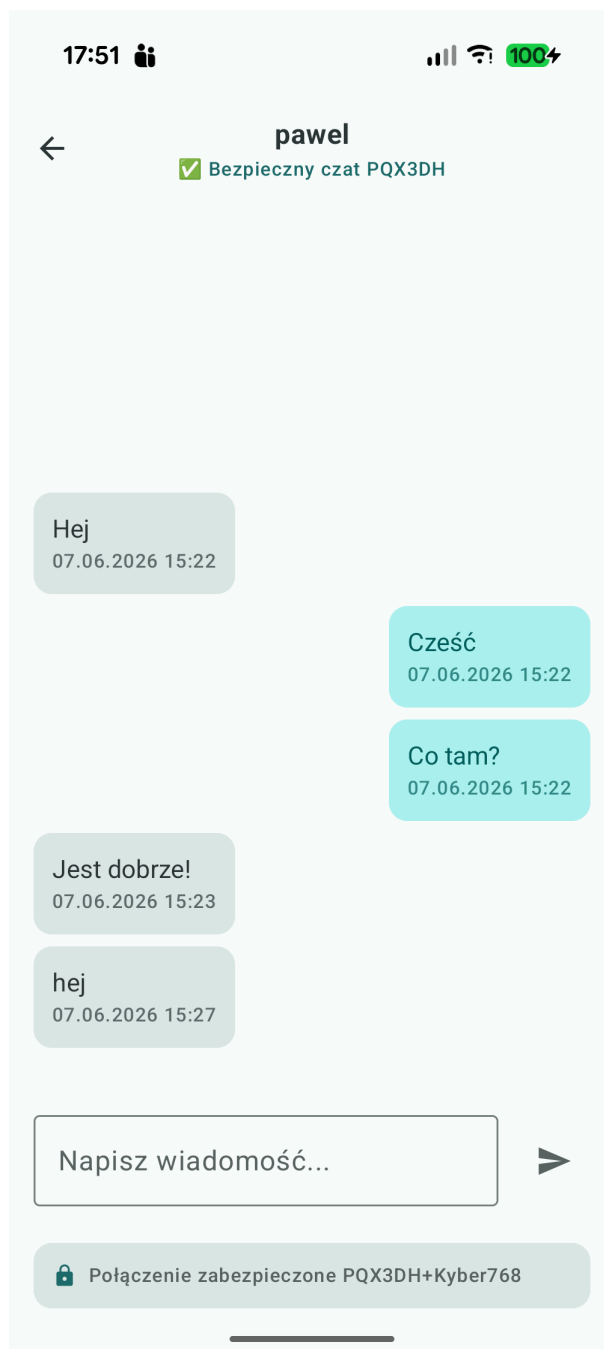
Rysunek 6.4: Ekran konfiguracji kluczy kryptograficznych – KeySetupScreen. Źródło: Opracowanie własne

Ekran **listy kontaktów** (UsersListScreen) wyświetla aktywnych i nieaktywnych użytkowników sieci. Pasek stanu połączenia informuje o aktualnym statusie połączenia z węzłem. Szuflada nawigacyjna (AppDrawer) udostępnia opcje odświeżania listy, ustawień oraz wylogowania. Przycisk nowej rozmowy (NewMessageScreen) umożliwia inicjowanie rozmowy z użytkownikiem przez podanie jego identyfikatora, nawet jeśli nie jest widoczny na liście. Rysunek 6.5 przedstawia wygląd ekranu listy kontaktów.



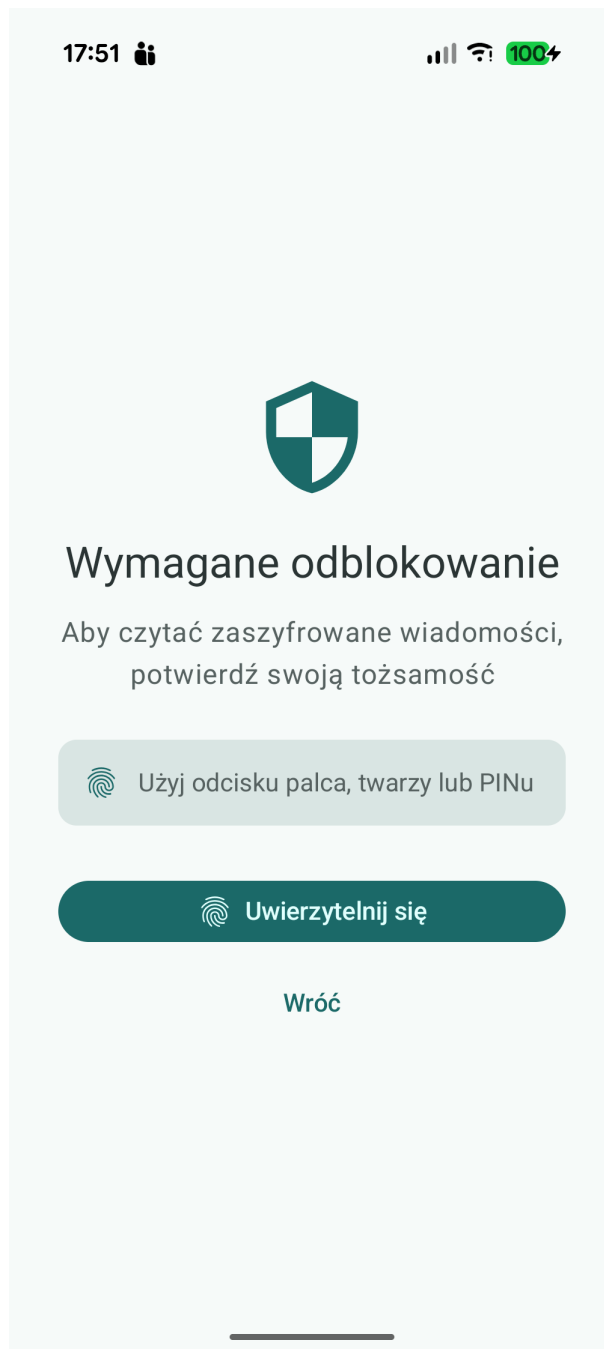
Rysunek 6.5: Ekran listy kontaktów – UsersListScreen. Źródło: Opracowanie własne

Ekran **czatu** (ChatScreenEnhanced) stanowi główny interfejs komunikacji. Wiadomości wyświetlane są w bąbelkach z rozróżnieniem nadawcy i odbiorcy. Przy pierwszym otwarciu rozmowy ekran sprawdza stan Magazynu kluczy Android – jeśli klucz główny wymaga uwierzytelnienia, użytkownik przekierowywany jest na ekran odblokowania przed wyświetleniem historii. Wysłanie wiadomości inicjuje procedurę PQX3DH, jeśli sesja kryptograficzna nie istnieje lub korzysta z istniejącego stanu podwójnej zapadki, jeśli sesja jest aktywna. Rysunek 6.6 przedstawia wygląd okna rozmowy.



Rysunek 6.6: Ekran rozmowy – ChatScreenEnhanced. Źródło: Opracowanie własne

Ekran **odblokowania** (UnlockScreen) wyświetlany jest, gdy Magazyn kluczy Android wymaga potwierdzenia tożsamości. Prezentuje monit biometryczny przez BiometricPrompt obsługujący odcisk palca, rozpoznanie twarzy i PIN urządzenia jako mechanizm zastępczy. Ekran wykrywa dostępność biometrii i informuje użytkownika o konieczności jej rejestracji, jeśli żadna metoda nie jest skonfigurowana. Rysunek 6.7 przedstawia wygląd ekranu odblokowania.



Rysunek 6.7: Ekran uwierzytelnienia biometrycznego – UnlockScreen. Źródło: Opracowanie własne

Integrację warstwy kryptograficznej z interfejsem użytkownika ilustruje listing 6.29, przedstawiający logikę sprawdzania stanu Magazynu kluczy Android przy otwarciu ekranu czatu.

Listing 6.29: Sprawdzanie stanu Keystore i warunkowe przekierowanie na ekran odblokowania w ChatScreenEnhanced

```
LaunchedEffect(Unit) {  
    keystoreStatus =  
        secureMessageManager.checkKeystoreStatus(activity)  
    when (keystoreStatus) {  
        KeystoreStatus.LOCKED -> showUnlockScreen = true  
        KeystoreStatus.ERROR ->  
            errorMessage = "Problem z zabezpieczeniem kluczy"  
        else -> { /* Keystore gotowy */ }  
    }  
}
```

```

}

if (showUnlockScreen) {
    UnlockScreen(
        onUnlocked = {
            showUnlockScreen = false
            keystoreStatus = KeystoreStatus.AVAILABLE
        },
        onSkip = { onBack() }
    )
    return
}

```

Efekt `LaunchedEffect(Unit)` uruchamiany jest jednorazowo przy pierwszym kompozycji ekranu. Jeśli Magazyn kluczy zwróci status `LOCKED`, flaga `showUnlockScreen` powoduje zastąpienie całego ekranu czatu ekranem `UnlockScreen`. Po pomyślnym uwierzytelnieniu flaga jest kasowana i ekran czatu staje się widoczny z pełną historią wiadomości.

6.4.3.2 Nawigacja i cykl życia

Graf nawigacji zdefiniowany w `AppEntryPoint` obejmuje cztery destynacje: `setName`, `keySetup/{userId}/{displayName}/{isIncognito}`, `usersList/{myId}` oraz `chat/{myId}/{peerId}`. Listing 6.30 przedstawia logikę wyznaczania ekranu startowego.

Listing 6.30: Wyznaczanie ekranu startowego w `AppEntryPoint`

```

val startDestination = when {
    startPeerId != null && myId != null -> {
        // uruchomienie z powiadomienia -- otworz czat bezpośrednio
        "chat/${myId}/${startPeerId}"
    }
    loggedInUserId != null -> {
        // zapisana sesja -- pomin rejestracje
        "usersList/${loggedInUserId}"
    }
    else -> {
        // pierwsze uruchomienie
        "setName"
    }
}

```

Cofanie z ekranu listy kontaktów jest zablokowane przez `BackHandler`, co zapobiega powrotowi do ekranu rejestracji po zalogowaniu. Wylogowanie czyści zapisaną sesję i usuwa cały stos nawigacji przez `popUpTo(0) { inclusive = true }`.

W tle działa usługa pierwszoplanowa `ChatService` z typem `dataSync`, która utrzymuje aktywne połączenie `WebSocket` z węzłem i obsługuje subskrypcję `onNewMessage` niezależnie od stanu cyklu życia aktywności. Dzięki temu wiadomości i powiadomienia systemowe docierają do użytkownika również gdy aplikacja nie jest widoczna na pierwszym planie.

6.5 Implementacja GraphQL

Warstwa GraphQL stanowi jedyny interfejs komunikacyjny między aplikacją mobilną a węzłem backendowym. Poniższe podrozdziały opisują konfigurację klienta Apollo Kotlin po stronie aplikacji mobilnej, zdefiniowane operacje oraz mechanizm subskrypcji `WebSocket` realizowany przez usługę `ChatService`.

6.5.1 Konfiguracja Apollo Kotlin

Klient GraphQL zbudowany jest na bibliotece Apollo Kotlin [31] w wersji 4.3.3. Wtyczka Gradle `com.apollographql.apollo` generuje w czasie kompilacji typowane klasy Kotlin na podstawie schematu serwera i zdefiniowanych operacji GraphQL. Listing 6.31 przedstawia konfigurację wtyczki w pliku `build.gradle.kts`.

Listing 6.31: Konfiguracja wtyczki Apollo Kotlin w build.gradle.kts

```
apollo {
    service("service") {
        packageName.set("com.pawelindus.magisterka_bm")
        introspection {
            endpointUrl.set("http://192.168.50.1:8080/graphql")
            schemaFile.set(
                file("src/main/graphql/schema.graphqls"))
        }
        generateKotlinModels.set(true)
    }
}
```

Blok introspection umożliwia pobranie schematu bezpośrednio z działającego węzła poleceniem Gradle `downloadServiceApolloSchemaFromIntrospection`. Wygenerowane klasy operacji – takie jak `SendMessageMutation`, `GetUserKeysQuery` czy `OnNewMessageSubscription` – są w pełni typowane, co eliminuje ręczne parsowanie odpowiedzi JSON.

Instancja `ApolloClient` tworzona jest z dwoma transportami: standardowym HTTP dla zapytań i mutacji oraz `WebSocketNetworkTransport` dla subskrypcji. Oba transporty wskazują na ten sam adres węzła, lecz różnią się schematem URL. Listing 6.32 przedstawia budowę klienta w `ChatService`.

Listing 6.32: Budowa klienta Apollo z transportem WebSocket w ChatService

```
private val apolloClient by lazy {
    ApolloClient.Builder()
        .serverUrl("http://$serverAddress:8080/graphql")
        .subscriptionNetworkTransport(
            WebSocketNetworkTransport.Builder()
                .serverUrl("ws://$serverAddress:8080/graphql")
                .build()
        )
        .build()
}
```

Użycie delegata `by lazy` zapewnia, że klient jest tworzony jednorazowo przy pierwszym dostępie, po tym jak adres serwera zostanie przekazany przez Intent do usługi.

6.5.2 Operacje GraphQL po stronie klienta

Operacje GraphQL zdefiniowane są w plikach `.graphql` w katalogu `src/main/graphql`. Na ich podstawie wtyczka Apollo generuje typowane klasy Kotlin w czasie kompilacji. Listing 6.33 przedstawia dwie kluczowe operacje: mutację wysyłania wiadomości oraz subskrypcję odbierania.

Listing 6.33: Operacje GraphQL po stronie klienta – mutacja i subskrypcja

```
# Mutacja wysylania wiadomosci
mutation SendMessageMutation(
    $from: String!, $to: String!
    $ciphertext: String!, $ratchetHeader: String!
    $iv: String!, $messageNumber: Int!
    $ephemeralKey: String # tylko pierwsza wiadomosc
    $kyberCiphertext: String # tylko pierwsza wiadomosc
) {
    sendMessage(from: $from, to: $to,
        ciphertext: $ciphertext,
        ratchetHeader: $ratchetHeader,
        iv: $iv, messageNumber: $messageNumber,
        ephemeralKey: $ephemeralKey,
        kyberCiphertext: $kyberCiphertext) {
        id from to timestamp
    }
}

# Subskrypcja odbierania wiadomosci
subscription OnNewMessageSubscription($to: String!) {
    onNewMessage(to: $to) {
```

```

        id from to ciphertext
        ratchetHeader iv messageNumber
        ephemeralKey kyberCiphertext
    }
}

```

Pola `ephemeralKey` i `kyberCiphertext` są opcjonalne – obecne wyłącznie w pierwszej wiadomości inicjującej sesję PQX3DH. Serwer przekazuje je transparentnie do odbiorcy bez interpretacji. Odbiorca na ich podstawie wykonuje uzgodnienie PQX3DH w metodzie `finalizeResponderSession` przed zapisem wiadomości do bazy Room.

6.5.3 Subskrypcja WebSocket w tle

Odbieranie wiadomości w czasie rzeczywistym realizowane jest przez usługę pierwszoplanową `ChatService` dziedziczącą po `LifecycleService`. Usługa uruchamiana jest z flagą `START_STICKY`, co gwarantuje jej wznowienie przez system po ewentualnym zabiciu procesu.

Przed otwarciem subskrypcji WebSocket usługa wykonuje testowe zapytanie HTTP do serwera z limitem czasowym 5 sekund. Dopiero po potwierdzeniu dostępności serwera uruchamiana jest właściwa subskrypcja. Listing 6.34 przedstawia fragment metody `startSubscription`.

Listing 6.34: Subskrypcja `OnNewMessage` przez WebSocket w `ChatService`

```

apolloClient
    .subscription(OnNewMessageSubscription(safeMyId))
    .toFlow()
    .collect { response ->
        lastActivityTimestamp = System.currentTimeMillis()
        val msg = response.data?.onNewMessage

        if (msg != null) {
            // pierwsza wiadomosc zawiera dane handshake PQX3DH
            if (msg.ephemeralKey != null
                && msg.kyberCiphertext != null) {
                finalizeResponderSession(
                    safeMyId = safeMyId,
                    senderId = msg.from,
                    ephemeralKey = msg.ephemeralKey,
                    kyberCiphertext = msg.kyberCiphertext
                )
            }
            saveMessageToDatabase(msg)
            if (!isAppInForeground(this@ChatService)) {
                showIncomingNotification(
                    msg.from, "Nowa zaszyfrowana wiadomosc")
            }
        }
    }
}

```

Odebranie wiadomości zawierającej pola `ephemeralKey` i `kyberCiphertext` sygnalizuje, że jest to pierwsza wiadomość w konwersacji wysłana przez inicjatora PQX3DH. Usługa wywołuje wówczas `finalizeResponderSession`, która wykonuje uzgodnienie po stronie odbiorcy i tworzy lokalną sesję kryptograficzną przed zapisaniem wiadomości do bazy Room. Dla kolejnych wiadomości pola te są puste i usługa przechodzi bezpośrednio do zapisu.

6.5.4 Niezawodność połączenia

Usługa implementuje dwa mechanizmy utrzymania połączenia. Jeden uruchamiany co 10 sekund sprawdza znacznik czasu ostatniej aktywności WebSocket. Jeśli od ostatniej odebranej ramki minęło więcej niż 20 sekund, subskrypcja uznawana jest za martwą i jest wznowiana. Drugi wysyła co 15 sekund testowe zapytanie `GetUserQuery` do serwera. Niepowodzenie tego drugiego inicjuje ponowne nawiązanie połączenia z opóźnieniem 5 sekund. Stan połączenia eksponowany jest jako `StateFlow<ConnectionStatus>` z czterema wartościami: `CONNECTED`, `CONNECTING`, `DISCONNECTED` i `ERROR`, subskrybowanym przez ekran listy kontaktów do wyświetlania paska stanu.

6.6 Narzędzia deweloperskie i pomocnicze

Niniejszy podrozdział opisuje dwa narzędzia pomocnicze zaimplementowane w trakcie prac deweloperskich, które usprawniły codzienny cykl pracy z węzłami Raspberry Pi. Podrozdział 6.6.1 omawia klasę `WifiAPManager` umożliwiającą programowe zarządzanie punktem dostępowym Wi-Fi, natomiast podrozdział 6.6.2 opisuje mechanizm ochrony przed wielokrotnym uruchomieniem aplikacji zaimplementowany w klasie `SingleInstanceGuard`.

6.6.1 Programowe zarządzanie punktem dostępowym

W trakcie prac deweloperskich wielokrotnie zachodziła potrzeba szybkiego resetowania lub ponownego uruchamiania punktu dostępowego Wi-Fi bez ręcznego wykonywania sekwencji poleceń przez SSH. Klasa `WifiAPManager` hermetyzuje kompletną procedurę zarządzania usługami `hostapd` i `dnsmasq` za pomocą metod `startAccessPoint` i `stopAccessPoint`, wywoływanych zarówno z wiersza poleceń, jak i z panelu administracyjnego.

Metoda `startAccessPoint` wykonuje sekwencję kroków w ściśle określonej kolejności, przedstawioną na listingu 6.35. Najpierw zatrzymywane są usługi `hostapd` i `dnsmasq` oraz `wpa_supplicant`, a zarządzanie interfejsem `wlan0` odbierane jest usłudze `NetworkManager`. Następnie interfejs otrzymuje statyczny adres IP, po czym uruchamiane są kolejno `dnsmasq` i `hostapd` z celowymi opóźnieniami między krokami. Opóźnienia są konieczne – zbyt szybkie uruchomienie `hostapd` przed zakończeniem konfiguracji interfejsu kończyło się błędem inicjalizacji. Po uruchomieniu usług metoda weryfikuje ich stan przez `systemctl is-active` i zwraca wartość logiczną informującą o powodzeniu operacji.

Listing 6.35: Metoda `startAccessPoint` – sekwencja uruchamiania punktu dostępowego Wi-Fi

```
public static boolean startAccessPoint() {
    try {
        // Krok 1: zatrzymaj wpa_supplicant i NetworkManager
        executeCommand("sudo", "systemctl", "stop",
            "hostapd", "dnsmasq");
        executeCommand("sudo", "systemctl", "stop",
            "wpa_supplicant@" + IFACE + ".service");
        executeCommand("sudo", "nmcli", "device",
            "set", "wlan0", "managed", "no");
        Thread.sleep(1000);

        // Krok 2: skonfiguruj interfejs
        executeCommand("sudo", "ip", "link", "set", IFACE, "up");
        executeCommand("sudo", "ip", "addr", "flush", "dev", IFACE);
        executeCommand("sudo", "ip", "addr", "add", AP_IP, "dev", IFACE);
        Thread.sleep(500);

        // Krok 3: uruchom dnsmasq
        executeCommand("sudo", "systemctl", "restart", "dnsmasq");
        Thread.sleep(1000);

        // Krok 4: uruchom hostapd
        executeCommand("sudo", "iw", "reg", "set", REGION);
        executeCommand("sudo", "systemctl", "restart", "hostapd");
        Thread.sleep(2000);

        // Weryfikacja statusu
        boolean hostapdOk = checkServiceStatus("hostapd");
        boolean dnsmasqOk = checkServiceStatus("dnsmasq");
        return hostapdOk && dnsmasqOk;

    } catch (Exception e) {
        System.err.println("[WiFi-AP] Bład: " + e.getMessage());
        return false;
    }
}
```

Dostępność tej funkcjonalności w panelu administracyjnym znacząco przyspieszyła cykl deweloperski – reset punktu dostępowego po zmianie konfiguracji `hostapd` sprowadzał się do kliknięcia przycisku w przeglądarce zamiast ręcznego wykonywania kilku poleceń przez SSH na każdym węźle osobno.

6.6.2 Ochrona przed wielokrotnym uruchomieniem

Środowisko deweloperskie na Raspberry Pi wielokrotnie prowadziło do sytuacji, w której poprzednia instancja aplikacji nie została poprawnie zakończona i nadal zajmowała port TCP serwera. Ponowna próba uruchomienia kończyła się wówczas wyjątkiem `BindException` rzucanym przez Spring Boot, wymagając ręcznego odnalezienia i zakończenia procesu przez `ps` i `kill`. Klasa `SingleInstanceGuard` automatyzuje tę procedurę.

Po wykryciu `BindException` klasa wyszukuje procesy pasujące do wzorca `thread-chat` narzędziem `pgrep`, wykluczając PID bieżącego procesu. Dla każdego znalezionej konfliktu wyświetlany jest interaktywny monit z pytaniem o zgodę na zakończenie procesu. Listing 6.36 przedstawia implementację obsługi konfliktu oraz zakończenia procesu.

Listing 6.36: Obsługa konfliktu portu i dwuetapowe kończenie procesu w `SingleInstanceGuard`

```
private static void handleBindException(
    BindException e, ThrowingRunnable startLogic) {
    try {
        List<Long> pids = findOtherInstancePids();

        if (pids.isEmpty()) {
            throw new RuntimeException(e);
        }

        for (long pid : pids) {
            if (askUser(pid)) {
                killProcess(pid);
                Thread.sleep(500);
                startLogic.run(); // ponow probe startu
                return;
            }
        }
        System.exit(0);
    } catch (Exception ex) {
        throw new RuntimeException(ex);
    }
}

private static void killProcess(long pid) {
    ProcessHandle.of(pid).ifPresent(ph -> {
        ph.destroy(); // SIGTERM
        try { Thread.sleep(500); }
        catch (InterruptedException ignored) {}
        if (ph.isAlive()) {
            ph.destroyForcibly(); // SIGKILL jesli nadal zyje
        }
    });
}
```

Zakończenie procesu przebiega dwuetapowo: najpierw wysyłany jest sygnał `SIGTERM` umożliwiający czysty shutdown aplikacji, a po 500 ms – jeśli proces nadal działa – sygnał `SIGKILL`. Po pomyślnym zakończeniu poprzedniej instancji procedura startowa jest ponawiana automatycznie przez wywołanie `startLogic.run()`. Interfejs funkcyjny `ThrowingRunnable` pozwala przekazać logikę startową jako wyrażenie lambda, co eliminuje duplikację kodu inicjalizującego w miejscu wywołania.

6.7 Problemy napotkane podczas implementacji

Niniejszy podrozdział opisuje najistotniejsze problemy napotkane podczas implementacji systemu oraz zastosowane rozwiązania.

6.7.1 Budowanie narzędzi OpenThread

Pierwszym problemem był brak gotowych pakietów binarnych OpenThread dla platformy Raspberry Pi, co wymuszało kompilację ze źródeł. Próba przeprowadzenia kompilacji na Raspberry Pi Zero 2W zakończyła się niepowodzeniem – czterokrotne uruchomienie procesu budowania kończyło się błędem braku pamięci operacyjnej, nawet po włączeniu pliku wymiany przez pakiet `dphys-swapfile`. Wynika to z ograniczeń sprzętowych modelu Zero 2W (512 MB RAM), niewystarczających dla procesu kompilacji projektu tej wielkości. Problem rozwiązano przez przeprowadzenie kompilacji na Raspberry Pi 4 i Raspberry Pi 5, a następnie osadzenie gotowych binarek ARM64 bezpośrednio w archiwum JAR aplikacji, co opisano w podrozdziale 6.2.

6.7.2 Adresowanie IPv6 w sieci Thread

Adresowanie węzłów w sieci Thread okazało się jednym z bardziej złożonych problemów implementacyjnych. Protokół Thread operuje na adresach IPv6 i przydziela każdemu węzłowi kilka rodzajów adresów o różnych właściwościach.

Pierwszym problemem było wymuszenie na gniazdach UDP korzystania z interfejsu `wpan0` zamiast domyślnego interfejsu sieciowego. System operacyjny domyślnie kierował ruch przez interfejs Wi-Fi, ignorując sieć Thread. Rozwiązaniem było jawne bindowanie gniazd do konkretnego adresu IPv6 interfejsu Thread zamiast do adresu ogólnego `:::`.

Drugim problemem było dobranie właściwego rodzaju adresu do komunikacji między węzłami. Adres RLOC (*Routing Locator*) okazał się funkcjonalny – pakiety wysyłane na aktualny RLOC węzła docierały do celu. Jego wadą jest jednak zmienność: adres RLOC może ulec zmianie przy rekonfiguracji topologii sieci, np. po wyborze nowego *Leader*a. Adres ML-EID (*Mesh-Local Endpoint Identifier*) jest teoretycznie stały dla danego urządzenia, jednak w praktyce nie wszystkie węzły rozpoznawały inne węzły po tym adresie i nie zawsze możliwe było wysyłanie pakietów na ML-EID innego węzła. Ostatecznym rozwiązaniem było połączenie obu mechanizmów: węzły identyfikowane są po stabilnym adresie MAC EUI64, natomiast aktualny adres RLOC wyznaczany jest dynamicznie na podstawie tablicy routerów odczytywanej z `ot-ctl` i odświeżanej co 30 sekund. Takie podejście zapewnia ciągłość komunikacji nawet przy zmianach topologii sieci.

6.7.3 Autonomia protokołu Thread

Protokół Thread zarządza tablicami routingu w sposób autonomiczny – po uruchomieniu węzeł musi wynegocjować role, adresy i trasy z pozostałymi uczestnikami sieci, co zajmuje od kilku do kilkunastu sekund. Próba komunikacji przez ThreadFluxBridge bezpośrednio po uruchomieniu węzła kończyła się pozornym brakiem działania – ramki były wysyłane, lecz nie docierały do celu z powodu braku skonfigurowanych tras. Problem rozwiązano przez wprowadzenie mechanizmu opóźnionej inicjalizacji ThreadFluxBridge oraz weryfikację stanu sieci przed pierwszą próbą komunikacji. W praktyce zaleca się odczekanie kilkunastu sekund po uruchomieniu wszystkich węzłów przed rozpoczęciem korzystania z warstwy aplikacyjnej.

6.7.4 Implementacja kryptografii na Androidzie

Implementacja protokołów PQX3DH i podwójnej zapadki po stronie aplikacji mobilnej okazała się znaczącym wyzwaniem implementacyjnym. Algorytmy kryptograficzne same w sobie są złożone matematycznie, a ich poprawna implementacja wymaga precyzyjnego przestrzegania kolejności operacji – nieprawidłowa kolejność konkatenacji sekretów DH lub błędna etykieta KDF skutkuje uzyskaniem różnych kluczy sesji po obu stronach, bez żadnego komunikatu o błędzie. Dodatkowym utrudnieniem był brak gotowych implementacji PQX3DH z Kyber dla platformy Android – protokół zaimplementowano od podstaw w oparciu o specyfikację Signal X3DH i dokumentację Bouncy Castle. Weryfikacja poprawności implementacji wymagała ręcznego porównywania pośrednich wartości kryptograficznych między urządzeniami testowymi.

6.7.5 Konfiguracja punktu dostępowego Wi-Fi

Konfiguracja interfejsu Wi-Fi na Raspberry Pi jako punktu dostępowego wymagała odebrania zarządzania interfejsem wlan0 usługą NetworkManager i skonfigurowania go ręcznie. Pominięcie tego kroku powodowało, że NetworkManager resetował konfigurację IP interfejsu, uniemożliwiając pracę serwera DHCP. Dodatkowym problemem była niekompatybilność modułów Wi-Fi Raspberry Pi z szyfrowaniem WPA3 – demon hostapd odmawiał uruchomienia punktu dostępowego w trybie WPA3 na testowanych modelach. Problem rozwiązano przez przejście na WPA2-PSK, które działa stabilnie na wszystkich testowanych węzłach.

Rozdział 7

Testy i badania

Opisywany poniżej rozdział przedstawia opracowanie wyników działania systemu w czterech obszarach: stabilności sieci Thread, niezawodności warstwy aplikacyjnej, poprawności kryptograficznej implementacji protokołów PQX3DH i podwójnej zapadki oraz wydajności operacji szyfrowania. Eksperymenty przeprowadzono na konfiguracji dwuwęzłowej z węzłami Raspberry Pi 4 i Raspberry Pi 5 z podłączonymi klientami mobilnymi. Uzyskane wyniki poddano syntetycznej analizie w sekcji 7.5.

7.1 Testy stabilności sieci Thread

Niniejszy podrozdział przedstawia wyniki testów weryfikujących stabilność i poprawność działania sieci Thread jako warstwy transportowej systemu. Testy przeprowadzono w konfiguracji dwuwęzłowej z węzłami Raspberry Pi 4 i Raspberry Pi 5. Do pomiaru czasów wykorzystano skrypty automatyczne wykrywające zmiany stanu węzłów oraz pomiar ręczny stoperem.

7.1.1 Formowanie sieci i czas gotowości

Scenariusz weryfikował czas od uruchomienia aplikacji backendowej na każdym węźle do momentu pełnej gotowości sieci Thread, rozumianej jako wzajemna widoczność węzłów i przydzielenie adresów RLOC.

Test przeprowadzono 4–5 razy. Węzły uruchamiano niemal jednocześnie z opóźnieniem nieprzekraczającym 1–2 sekund wynikającym z ręcznego uruchamiania kolejnych węzłów. Zmierzony czas od uruchomienia do wzajemnej widoczności węzłów wynosił od 30 do 45 sekund. Różnica między węzłami wynika z ich możliwości sprzętowych – Raspberry Pi 5 szybciej inicjalizuje wszystkie podprocesy i jest gotowy do komunikacji wcześniej niż Raspberry Pi 4, jednak oba węzły osiągały wzajemną widoczność w zbliżonym czasie, ponieważ sieć jest gotowa dopiero, gdy wszystkie węzły ukończą negocjację ról i adresów.

Wyniki potwierdzają obserwację opisaną w podrozdziale 6.6: po uruchomieniu węzłów należy oczekiwać około 30–45 sekund przed rozpoczęciem komunikacji przez warstwę aplikacyjną, aby protokół Thread zdążył wynegocjować topologię i przydzielić adresy RLOC.

7.1.2 Re-elekcja Lidera

Celem scenariusza była weryfikacja mechanizmu automatycznej re-elekcji Lidera sieci Thread po awarii węzła pełniącego tę rolę oraz wpływ re-elekcji na ciągłość adresowania w sieci.

Test przeprowadzono przez nagłe wyłączenie węzła pełniącego rolę Lidera i obserwację czasu do pojawienia się nowego Lidera na pozostałym węźle. Zmierzony czas re-elekcji wynosił 1 minutę i 45 sekund – tyle zajął protokołowi Thread wykrycie braku aktywności Lidera, przeprowadzenie nowej elekcji i przydzielenie roli pozostałemu węzłowi.

Tablica routerów na ocalałym węźle zaktualizowała się niezwłocznie po zakończeniu re-elekcji. Ponieważ routing wiadomości w systemie oparty jest na adresach RLOC odświeżanych z tablicy routerów, komunikacja powinna zostać wznowiona bez dodatkowych opóźnień po zakończeniu procesu re-elekcji. W czasie trwania re-elekcji wiadomości nie były wysyłane – scenariusz weryfikował wyłącznie mechanizm samego protokołu Thread, nie zaś ciągłość dostarczania wiadomości aplikacyjnych.

7.1.3 Długotrwała praca sieci

Stabilność systemu zweryfikowano przez ponad 7 dni nieprzerwanej pracy obu węzłów Raspberry Pi bez restartów i bez rozłączania połączenia między nimi. Do każdego węzła podłączony był jeden klient mobilny, co pozwalało weryfikować komunikację przez sieć Thread między węzłami.

W trakcie testu klienci mobilni wielokrotnie się rozłączali i ponownie podłączali – symulując rzeczywiste wzorce użytkowania. Węzły backendowe pozostawały nieprzerwanie uruchomione przez cały

czas trwania testu. Po każdym ponownym podłączeniu klienta mobilnego wiadomości były dostarczane poprawnie bez żadnych strat.

Przez cały czas trwania testu nie zaobserwowano ani jednej utraconej wiadomości. Wynik ten potwierdza stabilność zarówno warstwy transportowej Thread, jak i mechanizmu subskrypcji WebSocket z watchdogiem i heartbeatem opisanego w podrozdziale 6.5. Wysoka niezawodność przy tak długim czasie pracy jest wynikiem szczególnie pozytywnym, zważywszy na autonomiczny charakter protokołu Thread i dynamiczną naturę adresów RLOC.

Wyniki wszystkich trzech scenariuszy zestawiono w tabeli 7.1.

Tabela 7.1: Zestawienie wyników testów stabilności sieci Thread

Scenariusz	Metryka	Wynik
Formowanie sieci	Czas gotowości	30–45 s
Formowanie sieci	Liczba prób	4–5
Re-elekcja Leadera	Czas re-elekcji	1 min 45 s
Re-elekcja Leadera	Aktualizacja tablicy routerów	natychmiastowa
Soak test	Czas trwania	>7 dni
Soak test	Wskaźnik dostarczenia wiadomości	100%
Soak test	Straty wiadomości	0

7.2 Testy aplikacji mobilnej

Niniejszy podrozdział przedstawia wyniki testów weryfikujących niezawodność połączenia WebSocket, odporność na utratę łączności oraz poprawność działania usługi ChatService w tle. Testy przeprowadzono wielokrotnie – scenariusze 6.2.1 i 6.2.3 powtarzano ponad 10 razy przy każdym uruchomieniu aplikacji podczas prac deweloperskich.

7.2.1 Reconnect WebSocket po utracie połączenia Wi-Fi

Scenariusz weryfikował mechanizmy watchdog i heartbeat opisane w podrozdziale 6.5 oraz poprawność wyświetlania stanu `ConnectionStatus` w interfejsie użytkownika.

Zgodnie z implementacją watchdog sprawdza aktywność subskrypcji co 10 sekund. Jeśli od ostatniej odebranej ramki WebSocket minęło więcej niż 20 sekund, subskrypcja uznawana jest za martwą i inicjowany jest reconnect z opóźnieniem 1 sekundy. Maksymalny teoretyczny czas detekcji rozłączenia wynosi zatem do 30 sekund (10 s interwał watchdoga + 20 s próg braku aktywności). Heartbeat wysyła dodatkowe zapytanie `GetUserQuery` co 15 sekund z limitem czasowym 5 sekund – nieudany heartbeat również inicjuje reconnect.

Po przywróceniu połączenia aplikacja automatycznie wznowia subskrypcję i pobiera wiadomości zbuforowane podczas nieobecności z `MessageCache` serwera. Wiadomości wysłane przez rozmówcę podczas przerwy w połączeniu były dostarczane poprawnie po każdym powrocie do sieci. Test wykonywano wielokrotnie – scenariusz był weryfikowany przy każdym uruchomieniu aplikacji podczas prac deweloperskich, łącznie ponad 10 razy, za każdym razem z pozytywnym wynikiem.

7.2.2 Restart usługi ChatService

Scenariusz weryfikował działanie flagi `START_STICKY` – czy usługa automatycznie wznowia się po wymuszonym zatrzymaniu procesu i czy subskrypcja WebSocket jest ponownie otwierana.

Usługa ChatService zadeklarowana jest z flagą `START_STICKY`, co oznacza, że system Android automatycznie ją wznowia po zabiciu procesu. Po restarcie metoda `onStartCommand` ponownie inicjuje subskrypcję przez wywołanie `startSubscription`, która przed otwarciem WebSocket wykonuje testowe zapytanie HTTP z limitem 5 sekund w celu weryfikacji dostępności serwera. Całkowity czas od restartu usługi do aktywnej subskrypcji zależy od czasu odpowiedzi serwera i wynosi w typowym przypadku poniżej 10 sekund. Wiadomości odebrane przez serwer podczas okna bez aktywnej subskrypcji przechowywane są w `MessageCache` węzła i dostarczane po ponownym nawiązaniu subskrypcji.

7.2.3 Odporność na brak serwera przy starcie aplikacji

Scenariusz sprawdzał zachowanie aplikacji, gdy węzeł backendowy jest niedostępny w momencie uruchomienia.

Zgodnie z implementacją `startSubscription` wykonuje testowe zapytanie HTTP z limitem czasowym 5 sekund. Przy niedostępnym serwerze zapytanie kończy się wyjątkiem `TimeoutCancellationException` po upływie tego limitu, a stan połączenia przyjmuje wartość `ERROR`. Usługa automatycznie ponawia próbę połączenia co 5 sekund. Po uruchomieniu serwera heartbeat wysyłany co 15 sekund wykrywa jego dostępność i inicjuje pełną subskrypcję – maksymalny czas od uruchomienia serwera do stanu `CONNECTED` wynosi zatem do 20 sekund (5 s retry + 15 s heartbeat).

Aplikacja `reconnect`owała automatycznie bez żadnej interakcji użytkownika we wszystkich przeprowadzonych próbach, zgodnie z zaimplementowanym mechanizmem automatycznego wznawiania połączenia. Z logów diagnostycznych wynika, że od momentu uruchomienia serwera do uzyskania stanu `CONNECTED` minęło około 8 sekund – czas ten odpowiada pojedynczemu cyklowi retry (5 s opóźnienia) powiększonemu o czas nawiązania połączenia i potwierdzenia przez WebSocket.

Wartości parametrów mechanizmów niezawodności zastosowanych w `ChatService` zestawiono w tabeli 7.2.

Tabela 7.2: Parametry mechanizmów niezawodności `ChatService`

Mechanizm	Parametr	Wartość [s]
Watchdog	Interwał sprawdzania	10
Watchdog	Próg braku aktywności	20
Watchdog	Opóźnienie reconnect	1
Heartbeat	Interwał	15
Heartbeat	Limit czasowy zapytania	5
Reconnect	Czas od uruchomienia serwera do <code>CONNECTED</code>	≈8
Reconnect	Opóźnienie po błędzie	5
HTTP test	Limit czasowy	5

7.3 Testy poprawności szyfrowania

Niniejszy podrozdział przedstawia wyniki testów weryfikujących kryptograficzną poprawność implementacji protokołów PQX3DH i podwójnej zapadki opisanych w podrozdziale 6.4.2.

7.3.1 Weryfikacja uzgodnienia PQX3DH

Scenariusz potwierdzał, że inicjator i respondent wyprowadzają identyczny klucz sesji po pierwszej wymianie wiadomości. Weryfikację przeprowadzano przez porównanie wartości klucza sesji wypisywanego w logach diagnostycznych po obu stronach – po stronie inicjatora po wykonaniu `initiatorHandshake` oraz po stronie respondenta po wykonaniu `finalizeResponderSession`.

Test przeprowadzano wielokrotnie – od momentu ukończenia implementacji protokołu klucze sesji były identyczne w każdej próbie. Na wczesnym etapie implementacji zdarzały się rozbieżności wynikające z nieprawidłowej kolejności konkatenacji sekretów DH lub błędnej etykiety KDF, co opisano w podrozdziale 6.6. Po ustaleniu poprawnej implementacji test nigdy nie zakończył się niezgodnością kluczy.

7.3.2 Ciągłość sesji podwójnej zapadki

Scenariusz weryfikował poprawność rotacji kluczy w ramach algorytmu podwójnej zapadki przez długie sekwencje wiadomości. Testy przeprowadzano w formie organicznych konwersacji między dwoma urządzeniami, obejmujących wiadomości zawierające znaki specjalne i emotikony. Obecność znaków spoza podstawowego zakresu ASCII stanowi dodatkową weryfikację poprawności kodowania i dekodowania danych binarnych na wszystkich etapach przetwarzania.

Wszystkie wiadomości we wszystkich przeprowadzonych konwersacjach były poprawnie szyfrowane i odszyfrowywane. Nie zaobserwowano ani jednego błędu deszyfrowania przez cały okres testowania aplikacji.

7.3.3 Weryfikacja ekranu odblokowania Magazynu kluczy

Scenariusz potwierdzał, że bez uwierzytelnienia biometrycznego lub PIN klucze kryptograficzne są niedostępne i ekran czatu jest zablokowany.

Po upływie skonfigurowanego okresu ważności uwierzytelnienia (600 sekund, opisanego w podrozdziale 6.4.2) każda próba dostępu do historii wiadomości skutkuje wyświetleniem ekranu UnlockScreen. Ekran odblokowania pojawia się natychmiastowo – jest to systemowy monit BiometricPrompt wyświetlany przez Android bez opóźnień po stronie aplikacji.

Przed uwierzytelnieniem historia wiadomości była całkowicie niedostępna – nie było możliwości podglądu treści ani interakcji z ekranem czatu. Próba ominięcia okresu ważności klucza skutkuje brakiem możliwości odszyfrowania wiadomości, ponieważ Magazyn kluczy Android odmawia operacji kryptograficznych bez aktualnego uwierzytelnienia. Po pomyślnym uwierzytelnieniu pełna historia wiadomości była dostępna natychmiast.

Wyniki testów kryptograficznych zestawiono w tabeli 7.3.

Tabela 7.3: Zestawienie wyników testów poprawności szyfrowania

Scenariusz	Metryka	Wynik
uzgodnienie PQX3DH	Zgodność kluczy sesji	100% prób
uzgodnienie PQX3DH	Błędy po ustabilizowaniu implementacji	0
podwójna zapadka	Błędy deszyfrowania	0
podwójna zapadka	Obsługa znaków specjalnych i emotikon	poprawna
Magazyn kluczy	Historia niedostępna przed auth	tak
Magazyn kluczy	Czas wyświetlenia UnlockScreen	natychmiastowy
Magazyn kluczy	Dostęp po uwierzytelnieniu	pełny

7.4 Testy wydajnościowe

Niniejszy podrozdział przedstawia wyniki pomiarów wydajnościowych systemu, zrealizowanych zgodnie z sugestią promotora dotyczącą uwzględnienia aspektów wydajnościowych w badaniach.

7.4.1 Opóźnienie szyfrowania od końca do końca

Opóźnienie od końca do końca mierzono jako czas od wysłania wiadomości przez użytkownika do jej zapisu w bazie Room po stronie odbiorcy, na podstawie znaczników czasu z logów diagnostycznych aplikacji.

Pierwsza wiadomość w konwersacji, inicjująca uzgodnienie PQX3DH, charakteryzuje się wyższym opóźnieniem – w zmierzonym przypadku wyniosło ono około 2 sekund. Wynika to z konieczności wykonania po stronie odbiorcy pełnego uzgodnienia PQX3DH, który obejmuje pobranie kluczy publicznych inicjatora z serwera GraphQL, dekapsulację Kyber-768 i wyprowadzenie klucza sesji. Operacje te wykonywane są asynchronicznie przez usługę ChatService po odebraniu pierwszej wiadomości.

Dla kolejnych wiadomości w aktywnej sesji, gdzie uzgodnienie zostało już wykonane, zmierzone opóźnienia szyfrowania od końca do końca wyniosły od 96 ms do 15 ms w warunkach lokalnej sieci Wi-Fi. Wartości te obejmują czas przesłania wiadomości przez GraphQL WebSocket, przetworzenia przez serwer, dostarczenia przez subskrypcję do odbiorcy i odszyfrowania przez podwójną zapadkę.

7.4.2 Czas operacji kryptograficznych

Czasy operacji kryptograficznych zmierzono na podstawie znaczników czasu z logów diagnostycznych aplikacji działającej na urządzeniu testowym z systemem Android 15 (architektura ARM64).

Uzgodnienie PQX3DH po stronie inicjatora, obejmujący cztery wymiany X25519, enkapsulację Kyber-768 i wyprowadzenie klucza sesji funkcją KDF, zajął około 16 milisekund. Uzgodnienie po stronie respondenta, obejmujący dodatkowo pobranie kluczy z serwera, trwał około 28 milisekund – przy czym dominującą część tego czasu stanowiło zapytanie sieciowe, a nie same operacje kryptograficzne.

Szyfrowanie pojedynczej wiadomości przez podwójną zapadkę (krok KDF łańcucha i AES-256-GCM) zajmuje poniżej 1 milisekundy, co potwierdza brak wpływu na responsywność interfejsu użytkownika. Wszystkie operacje kryptograficzne wykonywane są w korutynach na dyspozytorze `Dispatchers.Default`, nie blokując wątku UI.

Wyniki pomiarów wydajnościowych zestawiono w tabeli 7.4.

Tabela 7.4: Zestawienie wyników testów wydajnościowych

Scenariusz	Metryka	Wynik
Opóźnienie od końca do końca	Pierwsza wiadomość	≈2 s
Opóźnienie od końca do końca	Kolejne wiadomości	96–115 ms
uzgadnianie PQX3DH	Czas po stronie inicjatora	≈16 ms
uzgadnianie PQX3DH	Czas po stronie respondenta	≈28 ms
podwójna zapadka	Szyfrowanie wiadomości	<1 ms
podwójna zapadka	Klucze zgodne po obu stronach	tak

Zaobserwowano również zjawisko *race condition* przy równoczesnej inicjalizacji łańcucha odbierania przez wiele wątków – objawiające się błędami `BAD_DECRYPT` dla wiadomości odczytywanych współbieżnie przy pierwszym otwarciu konwersacji. Błędy te nie powodują utraty danych – wiadomości są ponownie odczytywane przy kolejnej kompozycji ekranu i wyświetlane poprawnie.

7.5 Analiza wyników

Przeprowadzone testy obejmują cztery obszary: stabilność sieci Thread, niezawodność warstwy aplikacyjnej, poprawność kryptograficzną oraz wydajność systemu. Niniejszy podrozdział syntetyzuje uzyskane wyniki i wskazuje kierunki dalszego rozwoju.

7.5.1 Stabilność i niezawodność

Najważniejszym wynikiem przeprowadzonych badań jest ponad 7-dniowa nieprzerwana praca systemu bez utraty ani jednej wiadomości. Wynik ten potwierdza, że architektura oparta na połączeniu sieci rozproszonej Thread z reaktywnym potokiem Spring WebFlux i mechanizmem buforowania `MessageCache` jest wystarczająco odporna na typowe zakłócenia – rozłączenia klientów mobilnych, zmiany topologii sieci i wahania jakości połączenia Wi-Fi. Wskaźnik dostarczenia wiadomości na poziomie 100% przez cały okres testu wykracza poza zakładane wymagania i potwierdza zasadność przyjętych rozwiązań architektonicznych.

Zmierzony czas re-elekcji Lidera sieci Thread wynoszący 1 minutę i 45 sekund jest wartością wynikającą z wewnętrznych mechanizmów protokołu Thread i nie stanowi ograniczenia z punktu widzenia celu pracy. Istotniejszą obserwacją jest to, że aplikacja backendowa w pełni dostosowuje się do zmiany topologii – tablica routerów jest odświeżana natychmiast po zakończeniu re-elekcji, a komunikacja wznowiana bez ingerencji operatora. Czas re-elekcji mógłby zostać skrócony przez ręczne dostrojenie parametrów protokołu Thread, jednak wykracza to poza zakres niniejszej pracy.

7.5.2 Poprawność kryptograficzna

Drugim kluczowym wynikiem jest 100-procentowa zgodność kluczy sesji PQX3DH po obu stronach komunikacji we wszystkich przeprowadzonych próbach po ustabilizowaniu implementacji. Wynik ten potwierdza poprawność implementacji protokołu – zarówno kolejności konkatenacji sekretów DH, jak i etykiety funkcji KDF. Brak błędów deszyfrowania w protokole podwójnej zapadki przez cały okres

testowania, przy konwersacjach obejmujących znaki specjalne i emotikony, świadczy o kompletności implementacji na poziomie wystarczającym do praktycznego zastosowania.

Zaobserwowane sporadyczne błędy BAD_DECRYPT przy pierwszym otwarciu konwersacji są skutkiem *race condition* podczas równoczesnej inicjalizacji łańcucha odbierania przez wiele wątków. Zjawisko to nie powoduje utraty danych ani trwałych błędów – aplikacja ponawia próbę deszyfrowania przy kolejnej kompozycji ekranu i wyświetla wiadomości poprawnie. Nie zawsze również w ogóle występuje. Stanowi jednak obszar do poprawy w przyszłych iteracjach implementacji.

7.5.3 Wydajność

Zmierzone opóźnienia szyfrowania od końca do końca na poziomie 96–115 ms dla kolejnych wiadomości w aktywnej sesji są wartościami akceptowalnymi dla komunikatora działającego w lokalnej sieci Wi-Fi. Narzut kryptograficzny protokołu podwójnej zapadki wynoszący poniżej 1 ms na wiadomość jest pomijalne z punktu widzenia responsywności interfejsu użytkownika. Czas uzgadniania PQX3DH po stronie inicjatora wynoszący około 16 ms, mimo złożoności obliczeniowej (cztery wymiany X25519 i enkapsulacja Kyber-768), nie jest odczuwalny przez użytkownika i mieści się w granicach akceptowalnych dla jednorazowej operacji inicjalizacji sesji.

7.5.4 Kierunki dalszego rozwoju

Na podstawie przeprowadzonych testów można wskazać dwa główne kierunki dalszego rozwoju systemu.

Pierwszym jest skrócenie czasu uruchamiania węzła do stanu gotowości – obecne 30–45 sekund oczekiwania na formowanie sieci Thread wynika z autonomicznego charakteru protokołu i jest narzucone przez jego specyfikację. Potencjalną optymalizacją byłoby wdrożenie mechanizmu wstępnego sprawdzania stanu sieci i wcześniejszego udostępniania warstwy aplikacyjnej klientom mobilnym, z jednoczesnym buforowaniem wiadomości do czasu pełnej gotowości warstwy transportowej.

Drugim kierunkiem jest wzbogacenie mechanizmów samonaprawy połączenia. Obecna implementacja operuje na dwustanowym modelu – połączenie jest aktywne albo wymaga restartu z ustalonym interwałem. Bardziej zaawansowane rozwiązanie obejmowałoby obserwację stanów pośrednich protokołu Thread i WebSocket i reagowanie na nie w sposób dostosowany do konkretnego scenariusza – na przykład rozróżnienie między chwilową utratą zasięgu Wi-Fi a awarią węzła backendowego, i zastosowanie odmiennych strategii wznowienia w każdym z tych przypadków.

Rozdział 8

Zastosowania i możliwości rozwoju

Niniejszy rozdział omawia potencjalne zastosowania opracowanego systemu w różnych środowiskach operacyjnych, ocenia jego skalowalność i przenośność oraz wskazuje konkretne kierunki rozbudowy, które mogłyby zwiększyć jego użyteczność praktyczną.

8.1 Potencjalne zastosowania systemu

Zaprojektowany system komunikacji łączy cechy rzadko występujące jednocześnie w dostępnych rozwiązaniach komercyjnych: decentralizację, szyfrowanie od końca do końca odporne na komputery kwantowe oraz niezależność od zewnętrznej infrastruktury sieciowej. Zestaw tych właściwości otwiera szereg praktycznych zastosowań.

8.1.1 Komunikacja w środowiskach korporacyjnych

Organizacje przetwarzające dane wrażliwe – kancelarie prawne, firmy farmaceutyczne, instytucje finansowe czy działy badawczo-rozwojowe – coraz częściej poszukują rozwiązań komunikacyjnych, które nie wymagają przekazywania treści rozmów zewnętrznym dostawcom usług. Opisany system umożliwia wdrożenie komunikatora wyłącznie na własnej infrastrukturze sprzętowej, bez uzależnienia od zewnętrznych serwerów. Zastosowane szyfrowanie PQX3DH oraz protokół podwójnej zapadki gwarantują, że nawet administrator węzła infrastruktury nie ma dostępu do treści wymienianych wiadomości. Dodatkową zaletą jest możliwość audytu kodu źródłowego – brak zamkniętego oprogramowania eliminuje ryzyko ukrytych backdoorów.

8.1.2 Środowiska przemysłowe i Internet Rzeczy

Protokół Thread zaprojektowano z myślą o środowiskach IoT – niskie zużycie energii, praca w paśmie 2,4 GHz oraz natywna obsługa protokołu IPv6 czynią go naturalnym wyborem dla instalacji przemysłowych. System może pełnić rolę bezpiecznego kanału komunikacyjnego między operatorami maszyn, systemami sterowania a służbami utrzymania ruchu w fabrykach, magazynach lub instalacjach energetycznych, gdzie standardowa infrastruktura sieciowa bywa zawodna lub celowo odizolowana od publicznego Internetu. Sieć rozproszona pozwala na stopniowe rozszerzanie zasięgu przez dodawanie kolejnych węzłów bez rekonfiguracji istniejącej infrastruktury.

8.1.3 Komunikacja kryzysowa

Całkowita niezależność od zewnętrznego dostępu do Internetu czyni system szczególnie wartościowym w sytuacjach kryzysowych. Przy awarii infrastruktury telekomunikacyjnej – wywołanej klęską żywiołową, atakiem cybernetycznym lub zniszczeniem fizycznym – węzły oparte na mikrokomputerach Raspberry Pi tworzą autonomiczną sieć komunikacyjną dla służb ratowniczych, szpitali polowych lub centrów zarządzania kryzysowego. Sieć kratowa Thread zapewnia odporność na awarię pojedynczych węzłów przez automatyczną re-elekcję Lidera i wyznaczanie alternatywnych tras transmisji, co zostało potwierdzone eksperymentalnie w testach opisanych w podrozdziale 7.1.

8.1.4 Zastosowania wojskowe i służby specjalne

Środowiska militarne stawiają komunikacji wymagania niemożliwe do spełnienia przez komercyjne komunikatory: brak zależności od infrastruktury cywilnej, odporność na zakłócenia, poufność na poziomie kryptograficznym oraz możliwość pracy w środowiskach bez dostępu do Internetu. System oparty na protokole Thread może działać jako taktyczna sieć komunikacyjna na poziomie plutonu lub kompanii, w której każdy żołnierz dysponujący smartfonem z zainstalowaną aplikacją staje się uczestnikiem zaszyfrowanej sieci. Zastosowanie algorytmów postkwantowych – Kyber-768 w ramach protokołu PQX3DH

– zabezpiecza komunikację przed atakami z użyciem przyszłych komputerów kwantowych, co ma szczególne znaczenie dla danych wymagających długoterminowej poufności.

8.2 Uniwersalność i skalowalność rozwiązania

Opisywany system projektowano z założeniem skalowalności i przenośności, co przejawia się na kilku poziomach architektury.

Warstwa sieciowa oparta na protokole Thread obsługuje, zgodnie ze specyfikacją, do 250 routerów w jednej sieci [12]. Każdy węzeł routerowy może obsługiwać wielu klientów mobilnych równocześnie, a liczba węzłów i klientów ograniczona jest wyłącznie charakterystyką sprzętową urządzeń i zasięgiem radiowym. W testowanej konfiguracji dwuwęzłowej system działał stabilnie przez ponad 7 dni bez utraty ani jednej wiadomości, co sugeruje, że architektura jest odpowiednia do wdrożeń produkcyjnych przy zachowaniu podobnych parametrów na węzłach.

Warstwa aplikacyjna backendowa zrealizowana jest jako wykonywalny plik JAR zawierający wszystkie zależności, w tym skompilowane binaria OpenThread. Oznacza to, że węzeł można uruchomić na dowolnym urządzeniu z systemem Linux w wariantcie 64-bitowym na architekturze ARM64 bez instalowania dodatkowych pakietów. W praktyce oznacza to przenośność między urządzeniami klasy Raspberry Pi 4 i Raspberry Pi 5, a potencjalnie również innymi mikrokomputerami jednopłytkowymi opartymi na procesorach ARM.

Kod backendu pisany był z założeniem pracy wielowęzłowej bez centralnego koordynatora. Dodanie nowego węzła do sieci sprowadza się do uruchomienia aplikacji z odpowiednimi parametrami i dołączenia modułu radiowego Thread – węzeł automatycznie wykrywa pozostałych uczestników sieci i rejestruje się w ich strukturach rozgłoszeniowych. Nie jest wymagana żadna ręczna aktualizacja konfiguracji istniejących węzłów.

Aplikacja mobilna nie jest powiązana z konkretnym węzłem – użytkownik podaje adres IP węzła przy pierwszym uruchomieniu i może zmienić punkt dostępowy bez utraty historii rozmów przechowywanej lokalnie w bazie Room. Klucze kryptograficzne generowane są jednorazowo i nie są powiązane z węzłem, co umożliwia swobodną zmianę punktu dostępowego bez renegotiacji sesji z istniejącymi rozmówcami.

8.3 Możliwości rozbudowy systemu

Na podstawie doświadczeń zebranych podczas implementacji i testowania systemu można wskazać kilka konkretnych kierunków jego dalszego rozwoju.

8.3.1 Przesyłanie zaszyfrowanych plików i multimediów

Obecna implementacja obsługuje wyłącznie wiadomości tekstowe. Naturalnym rozszerzeniem byłoby dodanie możliwości przesyłania plików, zdjęć i nagrań głosowych w formie zaszyfrowanej. Plik przed wysłaniem byłby szyfrowany kluczem wyprowadzonym ze stanu protokołu podwójnej zapadki, a jego treść przesyłana jako binarne pole wiadomości GraphQL lub przez dedykowany punkt końcowy HTTP. Sieć Thread charakteryzuje się ograniczoną przepustowością (do ok. 250 kbps na węzeł), dlatego przesyłanie dużych plików wymagałoby implementacji mechanizmu fragmentacji i reasemblacji.

8.3.2 Zaawansowane mechanizmy samonaprawy połączenia

Obecna implementacja operuje na uproszczonym modelu – połączenie jest aktywne albo wymaga restartu z ustalonym interwałem. Bardziej zaawansowane rozwiązanie obejmowałoby obserwację stanów pośrednich protokołów Thread i WebSocket z reaktywnym reagowaniem na konkretne scenariusze awarii: chwilową utratę zasięgu Wi-Fi, awarię węzła backendowego lub zmianę topologii sieci Thread. Dla każdego scenariusza można by zdefiniować odrębną strategię wznowienia – na przykład natychmiastowy reconnect przy utracie Wi-Fi, lecz opóźniony wykładniczo przy niedostępności serwera.

8.3.3 Skrócenie czasu rozruchu sieci

Zmierzony czas od uruchomienia węzłów do pełnej gotowości sieci Thread wynosi 30–45 sekund, co wynika z autonomicznego charakteru protokołu i czasu negocjacji ról oraz adresów RLOC. Potencjalną optymalizacją byłoby wdrożenie mechanizmu wstępnego sprawdzania stanu sieci i wcześniejszego udostępniania warstwy aplikacyjnej klientom mobilnym, z jednoczesnym buforowaniem wiadomości do czasu pełnej gotowości warstwy transportowej. Pozwoliłoby to skrócić odczuwalne opóźnienie startowe z perspektywy użytkownika aplikacji mobilnej.

8.3.4 Uwierzytelnienie sprzętowe FIDO2

Obecny model bezpieczeństwa opiera się na uwierzytelnieniu biometrycznym lub PIN-em urządzenia przez Magazyn kluczy Android. Alternatywnym lub uzupełniającym mechanizmem byłoby uwierzytelnienie sprzętowe zgodne ze standardem FIDO2/WebAuthn, umożliwiające zastosowanie kluczy bezpieczeństwa, np. YubiKey, zamiast lub obok biometrii. Takie rozwiązanie byłoby szczególnie wartościowe w środowiskach wymagających silniejszego uwierzytelnienia wieloskładnikowego, np. w zastosowaniach militarnych lub korporacyjnych o podwyższonych wymaganiach bezpieczeństwa.

8.3.5 Formalna weryfikacja kryptograficzna

Poprawność implementacji protokołu PQX3DH została zweryfikowana empirycznie przez porównanie wartości kluczy sesji między urządzeniami testowymi. Bardziej rygorystyczne podejście wymagałoby formalnej weryfikacji protokołu przy użyciu narzędzi takich jak *ProVerif* lub *Tamarin Prover*. Narzędzia te umożliwiają automatyczne dowodzenie właściwości bezpieczeństwa – poufności, autentyczności i utajniania z wyprzedzeniem – w modelach symbolicznych, niezależnie od poprawności konkretnej implementacji. Weryfikacja taka stanowiłaby istotny wkład naukowy i mogłaby być przedmiotem osobnej pracy badawczej.

8.3.6 Dedykowane urządzenie końcowe oparte na mikrokontrolerze

Architektura systemu nie ogranicza się wyłącznie do smartfonów jako urządzeń klienckich. Naturalnym kierunkiem rozwoju byłoby opracowanie dedykowanego, autonomicznego urządzenia końcowego opartego na mikrokontrolerze, np. ESP32, wyposażonego w moduł Wi-Fi, niewielki wyświetlacz np. e-ink lub oled oraz ograniczony zestaw przycisków funkcyjnych. Takie urządzenie łączyłoby się z węzłem backendowym przez punkt dostępowy Wi-Fi – analogicznie do aplikacji mobilnej – i mogłoby pełnić rolę pagera zdolnego do odbierania wiadomości i wysyłania predefiniowanych odpowiedzi (*OK*, *Tak*, *Nie*) lub potwierdzenia odbioru w formie sygnału ACK inicjowanego przez użytkownika.

Kluczowym wyzwaniem implementacyjnym byłoby przeniesienie protokołu PQX3DH na platformę mikrokontrolerową – implementacja w języku C/C++ musiałaby korzystać z bibliotek kryptograficznych zoptymalizowanych pod kątem ograniczonych zasobów sprzętowych, takich jak *mbedTLS* lub dedykowane implementacje Kyber-768 dla systemów wbudowanych. Ograniczony pobór energii mikrokontrolera ESP32 w połączeniu z niewielkim akumulatorem umożliwiłaby wielodniową pracę autonomiczną, co czyni takie rozwiązanie szczególnie atrakcyjnym w scenariuszach terenowych lub w środowiskach o ograniczonej dostępności infrastruktury.

Rozdział 9

Podsumowanie i wnioski

Niniejszy rozdział stanowi zwięźczenie pracy i ma na celu syntetyczne ujęcie najważniejszych rezultatów uzyskanych w toku realizacji projektu, ocenę stopnia realizacji założonego celu pracy, a także przedstawienie wniosków końcowych wynikających z przeprowadzonych prac implementacyjnych i badawczych. W pierwszej części przedstawiono zestawienie kluczowych osiągnięć dotyczących zarówno warstwy komunikacyjnej opartej na sieci Thread, jak i zastosowanych mechanizmów kryptograficznych oraz bezpieczeństwa przechowywania danych. W dalszej części dokonano krytycznej oceny realizacji celu pracy, odnosząc się do napotkanych trudności technicznych oraz sposobu ich przezwyciężenia. Rozdział zamyka część poświęcona wnioskowi końcowemu oraz refleksji nad przebiegiem całego procesu projektowego.

9.1 Podsumowanie osiągniętych rezultatów

W ramach realizacji niniejszej pracy udało się osiągnąć wszystkie kluczowe założenia projektowe. Po pierwsze, zrealizowano integrację warstwy GPIO z narzędziem `ot-ctl`, dzięki czemu aplikacja napisana w języku Java jest w stanie komunikować się z modułem dongle Thread bezpośrednio z poziomu kodu wysokopoziomowego. Na tej podstawie zaimplementowano mechanizmy komunikacji pomiędzy poszczególnymi węzłami sieci oraz w ramach jednego węzła, przy czym z perspektywy użytkownika i aplikacji klienckiej oba te scenariusze są jednolite i nierozróżnialne – co znacząco upraszcza logikę aplikacji oraz doświadczenie użytkownika.

Istotnym osiągnięciem jest również zapewnienie pełnego szyfrowania całego ruchu pomiędzy klientami na zasadzie szyfrowania od końca do końca. Serwer w przyjętej architekturze nie bierze udziału w procesie szyfrowania i przechowuje jedynie klucze publiczne niezbędne do zainicjowania komunikacji, co minimalizuje poziom zaufania, jaki użytkownik musi posiadać wobec infrastruktury centralnej. Wdrożono przy tym wszystkie zaplanowane na etapie projektowania mechanizmy kryptograficzne, w tym wymianę kluczy w oparciu o krzywą X25519, postkwantowy algorytm Kyber, a także rotację kluczy efemerycznych realizowaną zgodnie ze schematem podwójnej zapadki. Bezpieczeństwo przechowywania materiału kryptograficznego zapewniono poprzez wykorzystanie Magazynu kluczy Android, dodatkowo zabezpieczonego dostępem opartym na biometrycznej autentykacji systemowej, co eliminuje możliwość nieautoryzowanego odczytu kluczy nawet w przypadku przejęcia urządzenia.

Najważniejszym wnioskiem płynącym z przeprowadzonych prac jest dowiedzenie, że możliwe jest zbudowanie zdecentralizowanego i w pełni bezpiecznego komunikatora opartego na szyfrowaniu od końca do końca, który jest niezależny architektonicznie od większości obecnie dostępnych na rynku rozwiązań. Otrzymane rezultaty pokazują ponadto, że sieć Thread – pomimo iż została zaprojektowana przede wszystkim z myślą o zastosowaniach IoT, a nie o komunikacji tekstowej między użytkownikami – może z powodzeniem stanowić fundament działającego komunikatora. W efekcie powstało rozwiązanie o charakterze nowatorskim, niedostępne wcześniej w takiej formie na rynku.

9.2 Ocena realizacji celu pracy

Założony cel pracy został zrealizowany, jednak sama implementacja okazała się znacznie trudniejsza niż przewidywano na etapie planowania projektu. Szczególnie istotnym wyzwaniem były problemy ze stabilnością sieci Thread, złożoność mechanizmów adresowania węzłów, a także niejasna i często niepełna dokumentacja techniczna, która stanowiła poważną przeszkodę podczas implementacji najniższej warstwy prototypu – odpowiedzialnej za samą komunikację radiową i zarządzanie topologią sieci. Równolegle do prac nad warstwą sieciową rozwijana była aplikacja kliencka na platformę Android, której proces tworzenia okazał się równie czasochłonny jak rozwiązywanie problemów związanych z siecią Thread. Mechanizmy szyfrowania nie zostały zaimplementowane jednorazowo w pełnym zakresie, lecz wprowadzane były iteracyjnie, kolejnymi etapami, z każdą iteracją zwiększając poziom bezpieczeństwa przetwarzania danych. W efekcie tego procesu finalna wersja aplikacji klienckiej, opracowana w wersji

prototypowej, osiągnęła docelowy poziom bezpieczeństwa założony na etapie planowania projektu i w pełni obsługuje wszystkie zaplanowane mechanizmy szyfrowania.

9.3 Wnioski końcowe

Realizacja niniejszej pracy stanowiła dla autorów przedsięwzięcie o znacznie większej złożoności, niż wynikało to z pierwotnych założeń projektowych. Konieczność jednoczesnej pracy na wielu poziomach systemu – od konfiguracji warstwy sprzętowej i sieci Thread, przez implementację serwera w technologiach Java oraz Spring, aż po opracowanie aplikacji mobilnej wraz z mechanizmami kryptografii postkwantowej – wymagała połączenia wiedzy z wielu obszarów informatyki oraz licznych iteracji i poprawek na każdym z etapów prac.

Pomimo napotkanych trudności, w tym problemów ze stabilnością i konfiguracją sieci rozproszonej, ostateczny efekt prac potwierdza, że przyjęte założenia projektowe były słuszne, a postawiony cel – możliwy do osiągnięcia. Doświadczenia zdobyte w trakcie realizacji projektu, obejmujące zarówno aspekty techniczne, jak i organizacyjne pracy zespołowej, należy uznać za istotną wartość dodaną niniejszej pracy, niezależnie od jej wyniku końcowego w postaci działającego prototypu.

Bibliografia

1. Meta Platforms, Inc. WhatsApp. Dostęp: 09.06.2026. URL: <https://www.whatsapp.com>.
2. Telegram FZ-LLC. Telegram Messenger. Dostęp: 09.06.2026. URL: <https://telegram.org>.
3. Meta Platforms, Inc. Messenger. Dostęp: 09.06.2026. URL: <https://www.messenger.com>.
4. Session Technology Foundation. Session – Private Messenger. Dostęp: 09.06.2026. URL: <https://getsession.org>.
5. Savoir-faire Linux. Jami. Dostęp: 09.06.2026. URL: <https://jami.net>.
6. Signal Foundation. Signal Messenger. Dostęp: 09.06.2026. URL: <https://signal.org>.
7. Overleaf. Overleaf – Online LaTeX Editor. Dostęp: 17.06.2026. URL: <https://www.overleaf.com>.
8. Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 3 edition, 2017. Dostęp: 09.06.2026.
9. Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys & Tutorials*, 7(2), 2005. Dostęp: 25.05.2026. URL: <https://ieeexplore.ieee.org/abstract/document/1610546>, doi:10.1109/COMST.2005.1610546.
10. Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5), 2016. Dostęp: 25.05.2026. URL: <https://ieeexplore.ieee.org/abstract/document/7488250>, doi:10.1109/JIOT.2016.2579198.
11. Alysha Helmrigh, Samuel Markolf, Rui Li, Thomaz Carvalhaes, Yeowon Kim, Emily Bondank, Mukunth Natarajan, Nasir Ahmad, and Mikhail Chester. Centralization and Decentralization for Resilient Infrastructure and Complexity. *Environmental Research: Infrastructure and Sustainability*, 1(2):021001, 2021. Dostęp: 09.06.2026. URL: <https://iopscience.iop.org/article/10.1088/2634-4505/ac0a4f>, doi:10.1088/2634-4505/ac0a4f.
12. Thread Group. Thread 1.4 Specification. Technical report, Thread Group, 2026. Dostęp: 25.05.2026. URL: <https://www.threadgroup.org/support>.
13. Google LLC. OpenThread – Open Source Implementation of Thread, 2024. Dostęp: 25.05.2026. URL: <https://openthread.io>.
14. IEEE. IEEE Standard for Low-Rate Wireless Networks – IEEE Std 802.15.4-2020. IEEE Standards Association, 2020. DOI: 10.1109/IEEESTD.2020.9144691.
15. Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol, 2016. Dostęp: 25.05.2026. URL: <https://signal.org/docs/specifications/x3dh/>.
16. Moxie Marlinspike and Trevor Perrin. The Double Ratchet Algorithm, 2016. Dostęp: 25.05.2026. URL: <https://signal.org/docs/specifications/doubleratchet/>.
17. WhatsApp Inc. WhatsApp Encryption Overview – Technical White Paper, 2016. Dostęp: 25.05.2026. URL: <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>.
18. Telegram Messenger LLP. MTProto Mobile Protocol. Dostęp: 25.05.2026. URL: <https://core.telegram.org/mtproto>.
19. Matrix.org Foundation. Matrix Specification – Client-Server API. Dostęp: 25.05.2026. URL: <https://spec.matrix.org/latest/>.
20. Google LLC. Room Persistence Library – Android Developers. Dostęp: 25.05.2026. URL: <https://developer.android.com/training/data-storage/room>.
21. Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems* (3rd Edition), 2017. Dostęp: 2026-06-04. URL: <https://www.distributed-systems.net/>.
22. William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 7th edition, 2015.
23. Signal Messenger LLC. Signal Technical Specifications and Documentation, 2026. Dostęp: 2026-06-04. URL: <https://signal.org/docs/>.

24. Oracle Corporation. Java SE Documentation. Dostęp: 13.06.2026. URL: <https://docs.oracle.com/en/java/javase/>.
25. VMware Inc. Spring Boot Reference Documentation. Dostęp: 25.05.2026. URL: <https://docs.spring.io/spring-boot/docs/current/reference/html/>.
26. VMware Inc. Spring WebFlux – Reactive Web Framework. Dostęp: 25.05.2026. URL: <https://docs.spring.io/spring-framework/reference/web/webflux.html>.
27. GraphQL Foundation. GraphQL Specification. Dostęp: 13.06.2026. URL: <https://graphql.org>.
28. Samuel Imolorhe and contributors. Altair GraphQL Client. Dostęp: 13.06.2026. URL: <https://altairgraphql.dev>.
29. JetBrains s.r.o. Kotlin Programming Language – Documentation. Dostęp: 13.06.2026. URL: <https://kotlinlang.org/docs/home.html>.
30. Google LLC. Jetpack Compose – Android Developers. Dostęp: 13.06.2026. URL: <https://developer.android.com/jetpack/compose>.
31. Apollo GraphQL Inc. Apollo Kotlin – GraphQL Client for Android and JVM. Dostęp: 25.05.2026. URL: <https://www.apollographql.com/docs/kotlin/>.
32. Legion of the Bouncy Castle Inc. Bouncy Castle Cryptography Library. Dostęp: 25.05.2026. URL: <https://www.bouncycastle.org>.
33. Google LLC. Android Keystore System – Android Developers. Dostęp: 13.06.2026. URL: <https://developer.android.com/privacy-and-security/keystore>.
34. Gradle Inc. Gradle User Manual. Dostęp: 13.06.2026. URL: <https://docs.gradle.org/current/userguide/userguide.html>.
35. Apache Software Foundation. Apache Maven Project. Dostęp: 13.06.2026. URL: <https://maven.apache.org>.
36. JetBrains s.r.o. IntelliJ IDEA – The Java and Kotlin IDE. Dostęp: 13.06.2026. URL: <https://www.jetbrains.com/idea/>.
37. Google LLC. Android Studio. Dostęp: 13.06.2026. URL: <https://developer.android.com/studio>.
38. Gitea Authors. Gitea – Git with a Cup of Tea. Dostęp: 13.06.2026. URL: <https://about.gitea.com>.

Spis Tabel

6.1	Główne zależności projektu backendowego	31
6.2	Główne zależności projektu aplikacji mobilnej	31
6.3	Parametry uruchomienia węzła backendowego	34
6.4	Typy ramek protokołu BridgeFrame i ich przeznaczenie	43
7.1	Zestawienie wyników testów stabilności sieci Thread	64
7.2	Parametry mechanizmów niezawodności ChatService	65
7.3	Zestawienie wyników testów poprawności szyfrowania	66
7.4	Zestawienie wyników testów wydajnościowych	67

Spis Rysunków

4.1	Zrzut ekranu z narzędzia Altair GraphQL Client z widocznym zapytaniem <code>getAllUsers</code> . Źródło: Opracowanie własne	13
4.2	Zrzut ekranu ze środowiska IntelliJ IDEA. Źródło: Opracowanie własne	16
4.3	Zrzut ekranu ze środowiska Android Studio. Źródło: Opracowanie własne	17
4.4	Zrzut ekranu z repozytorium Gitea. Źródło: Opracowanie własne	18
5.1	Ogólna architektura systemu komunikacji zdecentralizowanej. Źródło: Opracowanie własne	19
5.2	Scenariusze routingu wiadomości: rozmówcy na różnych węzłach (a) oraz na tym samym węzle (b). Źródło: Opracowanie własne	20
5.3	Schemat warstw backendu węzła. Źródło: Opracowanie własne	23
5.4	Uproszczony schemat architektury aplikacji Android. Źródło: Opracowanie własne . . .	25
5.5	Diagram encji modelu danych aplikacji Android. Źródło: Opracowanie własne	26
5.6	Diagram uzgodnienia klucza sesji w protokole PQX3DH. Źródło: Opracowanie własne .	29
6.1	Ekran logowania panelu administracyjnego. Źródło: Opracowanie własne	35
6.2	Widok główny panelu administracyjnego – stan węzła Thread, lista peerów i logi konsoli. Źródło: Opracowanie własne	36
6.3	Ekran rejestracji użytkownika – <code>SetNameScreen</code> . Źródło: Opracowanie własne	51
6.4	Ekran konfiguracji kluczy kryptograficznych – <code>KeySetupScreen</code> . Źródło: Opracowanie własne	52
6.5	Ekran listy kontaktów – <code>UsersListScreen</code> . Źródło: Opracowanie własne	53
6.6	Ekran rozmowy – <code>ChatScreenEnhanced</code> . Źródło: Opracowanie własne	54
6.7	Ekran uwierzytelnienia biometrycznego – <code>UnlockScreen</code> . Źródło: Opracowanie własne	55

Spis Listingów

6.1	Budowanie narzędzi OpenThread ze źródeł	31
6.2	Wypakowanie binarnych zależności OpenThread z archiwum JAR – klasa NativeBinaries	32
6.3	Uruchomienie demona ot-daemon	32
6.4	Metoda wykonująca polecenia ot-ctl	33
6.5	Procedura rozruchowa węzła – dołączenie lub utworzenie sieci Thread	33
6.6	Plik konfiguracyjny punktu dostępowego /etc/hostapd/hostapd.conf	33
6.7	Uruchomienie węzła backendowego	34
6.8	Klasa startowa Spring Boot – ThreadChatApplication	36
6.9	Konfiguracja serwera – application.properties	37
6.10	Schemat GraphQL – schema.graphqls	37
6.11	Routing wiadomości w mutacji sendMessage	38
6.12	Subskrypcja onNewMessage – scalanie strumieni lokalnych i sieciowych	39
6.13	Mutacja publishKeys – rejestracja użytkownika w sieci	39
6.14	Parametry konfiguracyjne węzła – klasa Config	40
6.15	Budowanie ramki protokołu MsgFrame	41
6.16	Wyznaczanie klucza sesji transportowej po wymianie HELLO	41
6.17	Wyznaczanie czasu docelowego jako średniej ważonej odpowiedzi sąsiadów	42
6.18	Struktura ramki protokołu BridgeFrame	43
6.19	Rejestracja lokalnego klienta w ThreadFluxBridge	44
6.20	Routing wiadomości w ThreadFluxBridge	44
6.21	Encja UserEntity – globalna baza użytkowników	45
6.22	Encja MessageEntity – struktura kryptograficzna wiadomości	46
6.23	Tworzenie izolowanej bazy danych dla pary rozmówców – klasa DatabaseProvider ...	46
6.24	Wybrane metody interfejsu MessageDao	47
6.25	Uzgodnienie PQX3DH po stronie inicjatora	47
6.26	Enkapsulacja i dekapulacja Kyber-768 w klasie CryptoUtils	48
6.27	Funkcja wyprowadzania klucza wiadomości z klucza łańcucha (kdfCK)	49
6.28	Konfiguracja klucza głównego w Android Keystore	49
6.29	Sprawdzanie stanu Keystore i warunkowe przekierowanie na ekran odblokowania w ChatScreenEnhanced	55
6.30	Wyznaczanie ekranu startowego w AppEntryPoint	56
6.31	Konfiguracja wtyczki Apollo Kotlin w build.gradle.kts	57
6.32	Budowa klienta Apollo z transportem WebSocket w ChatService	57
6.33	Operacje GraphQL po stronie klienta – mutacja i subskrypcja	57
6.34	Subskrypcja OnNewMessage przez WebSocket w ChatService	58
6.35	Metoda startAccessPoint – sekwencja uruchamiania punktu dostępowego Wi-Fi	59
6.36	Obsługa konfliktu portu i dwuetapowe kończenie procesu w SingleInstanceGuard ...	60