UNIWERSYTET WSB MERITO POZNAŃ

Wydział Finansów i Bankowości

Zdecentralizowany System Komunikacji P2P z Szyfrowaniem End-to-End w języku Rust

PROJEKT DYPLOMOWY

Poznań 2025

UNIWERSYTET WSB MERITO POZNAŃ

Spis treści

UNIWERSYTET WSB MERITO POZNAŃ

DANE PARTNERÓW

A1. Dane Promotora

Imię i nazwisko	Rafał Brodziak
Stopień / Tytuł naukowy	doktor inżynier
Data i podpis	

A2. Dane członków zespołu projektu

Imię i nazwisko	Lukrecja Pleskaczyńska
Kierunek studiów	Informatyka
Tryb studiów	Stacjonarne
Data i podpis	

UNIWERSYTET WSB MERITO POZNAŃ

ZAŁOŻENIA PROJEKTU

B1. Opis projektu

B1.1. Uzasadnienie wyboru tematu

Współczesna komunikacja internetowa w przeważającej mierze opiera się na scentralizowanych architekturach, w których dane oraz metadane[1] użytkowników są przechowywane na serwerach dostawców usług. Taki model stwarza istotne ryzyko nadzoru, cenzury oraz wycieku danych podczas naruszeń bezpieczeństwa. Centralizacja prowadzi również do powstania pojedynczych punktów awaryjnych[2], gdzie awaria jednego serwera może uniemożliwić komunikację dla wielu użytkowników. Ponadto, użytkownicy tracą kontrolę nad własnymi danymi, które są przetwarzane przez operatorów platform komunikacyjnych, co rodzi pytania o prywatność i zgodność z przepisami ochrony danych osobowych.

Niniejszy projekt stanowi alternatywę do tradycyjnych komunikatorów internetowych, oferując bezpieczną komunikację peer-to-peer[3] bez konieczności korzystania z centralnych serwerów. Dzięki temu system ogranicza konieczność zaufania do pojedynczego operatora infrastruktury oraz zmniejsza ryzyko scentralizowanej cenzury, zapewniając jednocześnie szyfrowaną wymianę wiadomości między uczestnikami komunikacji. W przeciwieństwie do rozwiązań wykorzystujących skomplikowane mechanizmy przechodzenia przez translację adresów sieciowych[4] , w niniejszym podejściu skupiono się

[1]Metadane to informacje opisujące inne dane, w kontekście komunikacji obejmują m.in. czas wysłania wiadomości, adres IP nadawcy i odbiorcy, rozmiar przesyłanych danych czy lokalizację geograficzną użytkowników. Analiza metadanych pozwala często odtworzyć sieć powiązań społecznych oraz wzorce aktywności użytkowników, nawet bez dostępu do treści wiadomości, co stanowi istotne ryzyko prywatności.

[2]Pojedynczy punkt awaryjny (ang. *single point of failure*, SPOF) oznacza element systemu, którego awaria powoduje niezdolność całego systemu do funkcjonowania. Pojęcie to jest szeroko omawiane w literaturze dotyczącej niezawodności systemów rozproszonych, por. Kleppmann, *Designing Data-Intensive Applications* [34].

[3]Komunikacja peer-to-peer (P2P) to model architektury sieciowej, w którym węzły (uczestnicy) komunikują się bezpośrednio ze sobą, bez pośrednictwa centralnego serwera. Każdy węzeł może pełnić zarówno rolę klienta, jak i serwera. Klasyczne omówienie architektury P2P znajduje się m.in. w: Tanenbaum, Wetherall, *Computer Networks* [31].

[4]NAT traversal (ang. *Network Address Translation traversal*) oznacza zestaw technik umożliwiających

na wdrożeniu systemu działającego w obrębie sieci lokalnej, co pozwala na uproszczenie architektury, lepszą czytelność kodu oraz skoncentrowanie się na kluczowych funkcjonalnościach, takich jak szyfrowanie typu end-to-end[5] i intuicyjny interfejs webowy.

Implementacja projektu opiera się na wykorzystaniu zaawansowanych technik kryptograficznych do zapewnienia poufności oraz integralności przesyłanych wiadomości, przechowywaniu danych w rozproszonej tablicy haszującej[6] oraz automatycznym wykrywaniu węzłów w sieci lokalnej za pomocą protokołu mDNS[7]. Wybór języka programowania Rust wynika z jego wysokiej efektywności, gwarancji bezpieczeństwa pamięci oraz rozwiniętego ekosystemu bibliotek kryptograficznych i sieciowych. W projekcie wykorzystano między innymi bibliotekę libp2p – modułowy stos protokołów umożliwiający tworzenie aplikacji typu peer-to-peer, którego dokumentację można znaleźć na stronie internetowej projektu.

B1.2. Problem badawczy

W jaki sposób zaprojektować i zaimplementować zdecentralizowany system komunikacji peer-to-peer (bez centralnego serwera aplikacyjnego), który zwiększy poziom prywatności użytkowników poprzez zastosowanie szyfrowania end-to-end, umożliwi

nawiązywanie bezpośrednich połączeń P2P między urządzeniami znajdującymi się za routerami wykonującymi translację adresów sieciowych (NAT). Popularne rozwiązania obejmują protokoły STUN i TURN wykorzystywane m.in. w WebRTC [8, 9].

[5]Szyfrowanie end-to-end (E2EE) to metoda komunikacji, w której jedynie komunikujące się strony mogą odczytać wiadomości. Dane są szyfrowane na urządzeniu nadawcy i deszyfrowane dopiero na urządzeniu odbiorcy, uniemożliwiając ich odczyt przez pośredników, w tym dostawców usług. Przykładowym standardem wykorzystującym E2EE jest protokół Signal (por. Marlinspike, Perrin, *The Double Ratchet Algorithm* [27]).

[6]Rozproszona tablica haszująca (ang. *Distributed Hash Table*, DHT) to zdecentralizowana struktura danych rozłożona pomiędzy wieloma węzłami sieci, umożliwiająca efektywne przechowywanie i wyszukiwanie danych w systemach P2P bez potrzeby utrzymywania centralnego serwera. Przykładowym algorytmem DHT jest Kademlia, por. Maymounkov, Mazières, *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric* [14].

[7]Multicast DNS (mDNS) to protokół umożliwiający rozwiązywanie nazw hostów w małych sieciach lokalnych bez konieczności konfiguracji centralnego serwera DNS. Zdefiniowany w dokumencie RFC 6762, ujętym w bibliografii niniejszej pracy [3].

asynchroniczną wymianę wiadomości (w tym dostarczanie wiadomości do nieaktywnych użytkowników), a jednocześnie zapewni efektywne wykrywanie oraz łączenie się z innymi urządzeniami w obrębie sieci lokalnej przy użyciu mDNS?

B1.3. Cel główny i cele szczegółowe projektu

Realizacja głównego problemu badawczego wymaga rozwiązania szeregu zagadnień o charakterze technicznym i projektowym. Pierwszym z nich jest zapewnienie efektywnego wykrywania oraz łączenia się z innymi węzłami sieci w warunkach sieci lokalnej, co wiąże się z koniecznością wyboru odpowiedniego protokołu rozgłaszania obecności węzłów oraz mechanizmu automatycznego nawiązywania połączeń bez konieczności manualnej konfiguracji przez użytkowników. Kolejnym wyzwaniem jest implementacja bezpiecznego przechowywania wiadomości przeznaczonych dla nieaktywnych użytkowników w rozproszonej tablicy haszującej przy jednoczesnym zachowaniu pełnej poufności ich treści, co wymaga zastosowania odpowiednich technik kryptograficznych uniemożliwiających odczytanie zawartości przez węzły przechowujące dane. Trzecim istotnym problemem jest zaprojektowanie mechanizmu weryfikacji dostarczenia wiadomości, który umożliwi bezpieczne usuwanie danych z DHT po ich odczytaniu przez właściwego odbiorcę, jednocześnie zapobiegając nieautoryzowanemu usuwaniu wiadomości przez podmioty niepowołane. Wreszcie, konieczne jest zapewnienie autentyczności oraz integralności przesyłanych danych przy jednoczesnym zachowaniu anonimowości użytkowników, co wymaga starannego wyważenia pomiędzy wymogami bezpieczeństwa a ochroną prywatności uczestników komunikacji.

Cel główny: Stworzenie bezpiecznego, zdecentralizowanego komunikatora peer-to-peer, zaimplementowanego w języku Rust, wykorzystującego biblioteki takie jak **libp2p** oraz mechanizmy DHT do przechowywania historii wiadomości, a także zapewniającego intuicyjny interfejs użytkownika oparty na frameworku Vue.js.

Cele szczegółowe: Realizacja celu głównego projektu opiera się na osiągnięciu następujących celów cząstkowych:

1. **Analiza technologii i narzędzi (libp2p, szyfrowanie)** – Opracowanie architektury systemu peer-to-peer wykorzystującej bibliotekę libp2p, która zapewni modu-

larność oraz rozszerzalność rozwiązania. Przeprowadzenie analizy mechanizmów kryptograficznych, w tym algorytmu wymiany kluczy X25519[8] oraz szyfru strumieniowego ChaCha20-Poly1305[9], które będą stanowić podstawę szyfrowania end-to-end gwarantującego poufność komunikacji.

2. **Implementacja podstawowych funkcji (mDNS, wysyłanie wiadomości, historia, szyfrowanie)** – Wdrożenie mechanizmu automatycznego wykrywania węzłów w sieci lokalnej z wykorzystaniem protokołu mDNS wraz z możliwością przypisywania przez użytkowników pseudonimów ułatwiających identyfikację kontaktów. Implementacja mechanizmu szyfrowania end-to-end opartego na algorytmie wymiany kluczy X25519 oraz szyfru strumieniowego ChaCha20-Poly1305. Stworzenie funkcjonalności umożliwiającej wysyłanie wiadomości tekstowych oraz bezpieczne przechowywanie historii konwersacji w lokalnej bazie danych.

3. **Analiza wykonywalności DHT do przechowywania offline** – Integracja rozproszonego przechowywania wiadomości przeznaczonych dla nieobecnych użytkowników z wykorzystaniem mechanizmu DHT, co pozwoli na asynchroniczną wymianę informacji. Opracowanie mechanizmu weryfikacji dostarczenia wiadomości oraz autoryzowanego usuwania danych z węzłów przechowujących, co zapewni efektywne zarządzanie zasobami systemowymi.

4. **Projekt i implementacja interfejsu użytkownika** – Zaprojektowanie oraz implementacja responsywnego interfejsu użytkownika wykorzystującego framework Vue.js, umożliwiającego intuicyjną komunikację oraz zarządzanie kluczami kryp-

[8]X25519 to funkcja uzgadniania kluczy oparta na krzywej eliptycznej Curve25519, zaprojektowana przez Daniela J. Bernsteina, oferująca wysoki poziom bezpieczeństwa przy zachowaniu efektywności obliczeniowej. Schemat ten został znormalizowany w dokumencie RFC 7748 (Langley i in., *Elliptic Curves for Security*) [1].

[9]ChaCha20-Poly1305 to algorytm szyfrowania uwierzytelnionego (AEAD) łączący szyfr strumieniowy ChaCha20 z kodem uwierzytelniania wiadomości Poly1305, zapewniający zarówno poufność, jak i integralność danych. Zestaw ten został znormalizowany w dokumencie RFC 8439 (Nir, Langley, *ChaCha20 and Poly1305 for IETF Protocols*) [2].

tograficznymi.

5. **Testowanie i optymalizacja** – Zapewnienie skalowalności systemu poprzez implementację mechanizmów przeciwdziałających przeciążeniom sieci, takich jak wykładnicze wycofywanie przy ponownych próbach połączeń. Przeprowadzenie kompleksowych testów bezpieczeństwa oraz optymalizacji wydajnościowych systemu.

B1.4. Zakres podmiotowy, przedmiotowy, czasowy i przestrzenny

Zakres podmiotowy. Projekt skierowany jest do różnorodnych grup użytkowników wymagających bezpiecznych narzędzi komunikacyjnych. Do głównych adresatów rozwiązania należą osoby ceniące wysoką prywatność w komunikacji oraz poszukujące alternatywy dla komercyjnych platform scentralizowanych. Istotną grupą docelową są również aktywiści oraz dziennikarze śledczy działający w warunkach ograniczonej wolności słowa, dla których odporność na cenzurę oraz ochrona przed nadzorem stanowią kluczowe wymagania. Rozwiązanie może znaleźć zastosowanie w organizacjach przetwarzających wrażliwe dane, które ze względów bezpieczeństwa oraz zgodności z przepisami o ochronie danych osobowych muszą zapewnić maksymalną poufność komunikacji wewnętrznej. Ponadto, projekt adresowany jest do społeczności technologicznych zainteresowanych rozwiązaniami zdecentralizowanymi oraz do badaczy i entuzjastów technologii peer-to-peer oraz kryptografii, którzy mogą wykorzystać system jako platformę do dalszych eksperymentów i rozwoju.

Zakres przedmiotowy. Zakres rzeczowy projektu obejmuje kompleksową analizę istniejących rozwiązań wykorzystujących architekturę peer-to-peer oraz techniki szyfrowania end-to-end, co pozwala na identyfikację najlepszych praktyk oraz potencjalnych pułapek projektowych. Następnie projekt koncentruje się na zaprojektowaniu architektury systemu z wykorzystaniem biblioteki libp2p, która stanowi fundament warstwy sieciowej aplikacji. W zakresie prac implementacyjnych mieszczą się zarówno mechanizmy szyfrowania end-to-end zapewniające poufność komunikacji, jak i protokoły automatycznego wykrywania węzłów w sieci lokalnej przy użyciu mDNS. Projekt obejmuje również opracowanie i wdrożenie funkcjonalności umożliwiających wysyłanie wiadomości tekstowych między użytkownikami oraz bezpieczne przechowywanie historii konwersacji w lokalnej

bazie danych. Istotnym elementem jest integracja mechanizmu rozproszonej tablicy haszującej służącej do przechowywania wiadomości przeznaczonych dla użytkowników tymczasowo nieobecnych w sieci. Ponadto, realizacja obejmuje opracowanie mechanizmów weryfikacji dostarczenia wiadomości oraz autoryzowanego usuwania danych z węzłów przechowujących. Wreszcie, w zakresie projektu znajduje się zaprojektowanie oraz implementacja intuicyjnego interfejsu użytkownika wraz z funkcjonalnościami zarządzania kluczami kryptograficznymi, a także przeprowadzenie testów bezpieczeństwa oraz pomiarów wydajnościowych systemu.

Zakres czasowy. Realizacja projektu rozłożona jest na okres dwunastu miesięcy począwszy od grudnia roku 2024. Harmonogram prac przewiduje w pierwszych dwóch miesiącach (grudzień 2024 – styczeń 2025) przeprowadzenie analizy wymagań funkcjonalnych oraz niefunkcjonalnych, a także zaprojektowanie ogólnej architektury systemu. W kolejnych trzech miesiącach (luty – kwiecień 2025) planowana jest implementacja kluczowych funkcjonalności, obejmujących mechanizm wykrywania węzłów w sieci lokalnej przy użyciu protokołu mDNS, funkcje wysyłania wiadomości tekstowych, przechowywanie historii konwersacji oraz wdrożenie szyfrowania end-to-end. Miesiące od maja do lipca 2025 roku przeznaczone są na integrację mechanizmu rozproszonej tablicy haszującej służącego do przechowywania wiadomości przeznaczonych dla nieaktywnych użytkowników oraz opracowanie systemu weryfikacji dostarczenia wiadomości. W okresie od sierpnia do września 2025 roku zaplanowano zaprojektowanie oraz implementację responsywnego interfejsu użytkownika opartego na frameworku Vue.js. Dwa ostatnie miesiące projektu (październik – listopad 2025) przewidziane są na przeprowadzenie kompleksowych testów bezpieczeństwa, audytu kodu oraz optymalizacji wydajnościowej systemu.

Zakres przestrzenny. Ze względu na cel edukacyjny oraz badawczy projektu, wdrożenie systemu ogranicza się do środowiska sieci lokalnej. Takie podejście umożliwia skupienie uwagi na kluczowych aspektach bezpieczeństwa komunikacji oraz mechanizmach kryptograficznych, jednocześnie upraszczając proces prototypowania i testowania. Rezygnacja z implementacji zaawansowanych mechanizmów przechodzenia przez translację adresów sieciowych pozwala na bardziej przejrzystą architekturę systemu oraz uła-

twia weryfikację poprawności działania poszczególnych komponentów w kontrolowanym środowisku testowym.

B1.5. Metody i techniki badawcze

W projekcie zastosowano zestaw metod i technik badawczych właściwych dla prac o charakterze wdrożeniowym w dziedzinie inżynierii oprogramowania. Proces realizacji rozpoczęto od przeglądu literatury oraz istniejących rozwiązań z zakresu systemów peer-to-peer i protokołów kryptograficznych, co pozwoliło na wstępną identyfikację dobrych praktyk projektowych oraz typowych zagrożeń bezpieczeństwa. W fazie implementacji przyjęto metodykę prototypowania iteracyjnego, w ramach której kolejne funkcjonalności systemu były wdrażane w krótkich cyklach, co umożliwiało bieżące weryfikowanie założeń projektowych oraz reagowanie na napotkane problemy techniczne. Weryfikację poprawności podstawowych mechanizmów kryptograficznych prowadzono w sposób praktyczny, poprzez testowanie scenariuszy wymiany kluczy i szyfrowania wiadomości oraz sprawdzanie, czy dane przesyłane w sieci są nieczytelne dla podglądu narzędziem typu Wireshark. Interfejs użytkownika oceniano na bieżąco pod kątem intuicyjności obsługi podczas pracy z prototypem systemu. Dodatkowe, bardziej systematyczne testy wydajnościowe (m.in. pomiar opóźnień w dostarczaniu wiadomości, zużycia zasobów systemowych oraz zachowania systemu pod obciążeniem) są planowane na końcowym etapie realizacji projektu.

B2. Zadania w projekcie

Cele szczegółowe projektu	**Zadania i termin realizacji**	**Osoby zaangażowane**
Cel 1: Analiza technologii i narzędzi (libp2p, szyfrowanie)	Zadanie 1: Analiza rozwiązań P2P i biblioteki **libp2p** (grudzień 2024)	Lukrecja Pleskaczyńska

Cele szczegółowe projektu	Zadania i termin realizacji	Osoby zaangażowane
	Zadanie 2: Analiza kryptografii (X25519, ChaCha20-Poly1305) (styczeń 2025)	Lukrecja Pleskaczyńska
Cel 2: Implementacja podstawowych funkcji (mDNS, wysyłanie wiadomości, historia, szyfrowanie)	Zadanie 1: Wdrożenie wykrywania węzłów w sieci lokalnej (mDNS) i przesyłania wiadomości (luty 2025)	Lukrecja Pleskaczyńska
	Zadanie 2: Implementacja historii czatu i zarządzania kluczami (marzec 2025)	Lukrecja Pleskaczyńska
	Zadanie 3: Szyfrowanie end-to-end (X25519 + ChaCha20-Poly1305) oraz testy (kwiecień–maj 2025)	Lukrecja Pleskaczyńska
Cel 3: Analiza wykonywalności DHT do przechowywania offline	Zadanie 1: Projekt mechanizmu DHT i podstawowa integracja (maj–czerwiec 2025, ograniczony zakres)	Lukrecja Pleskaczyńska
	Zadanie 2: Testy funkcjonalności DHT i weryfikacja dostarczenia (lipiec 2025)	Lukrecja Pleskaczyńska
Cel 4: Projekt i implementacja interfejsu użytkownika	Zadanie 1: Projekt UI i makiety (sierpień 2025)	Lukrecja Pleskaczyńska
	Zadanie 2: Implementacja frontendowa (Vue.js) (wrzesień 2025)	Lukrecja Pleskaczyńska

Cele szczegółowe projektu	Zadania i termin realizacji	Osoby zaangażowane
	Zadanie 3: Integracja UI z backendem (październik 2025)	Lukrecja Pleskaczyńska
Cel 5: Testowanie i optymalizacja	Zadanie 1: Testy penetracyjne i audyt bezpieczeństwa (październik 2025)	Lukrecja Pleskaczyńska
	Zadanie 2: Testy wydajnościowe i symulacje przeciążeń (październik 2025)	Lukrecja Pleskaczyńska
	Zadanie 3: Wprowadzenie poprawek i optymalizacja (listopad 2025)	Lukrecja Pleskaczyńska

C1. Opracowanie projektu

C1.1. Założenia teoretyczne

C1.1.1. Wprowadzenie do komunikacji peer-to-peer

Architektura systemów sieciowych przeszła znaczącą ewolucję od momentu powstania sieci ARPANET w latach 60. XX wieku. Pierwotne modele komunikacji internetowej opierały się na bezpośredniej wymianie danych między równorzędnymi węzłami, co odpowiadało koncepcji sieci rozproszonych. Wraz z komercjalizacją internetu w latach 90. XX wieku dominującym paradygmatem stał się jednak model klient-serwer, w którym centralne serwery pełnią rolę pośredników w komunikacji oraz przechowują dane użytkowników. Taki model, choć skuteczny z perspektywy zarządzania zasobami oraz skalowania usług, wprowadził istotne ograniczenia związane z centralizacją kontroli oraz zwiększonym ryzykiem awarii systemowych.

Model klient-serwer charakteryzuje się wyraźnym podziałem ról pomiędzy dwie kategorie uczestników sieci. Serwery stanowią węzły o wysokiej dostępności, które udostępniają zasoby oraz świadczą usługi, natomiast klienci są odbiorcami tych usług, inicjującymi połączenia oraz przesyłającymi żądania do serwerów. Taka asymetria ról prowadzi do skoncentrowania danych oraz logiki przetwarzania w centralnych lokalizacjach, co stwarza pojedyncze punkty awaryjne oraz umożliwia stosunkowo prostą kontrolę przepływu informacji przez operatorów infrastruktury. Ponadto, model ten wymaga znaczących nakładów infrastrukturalnych na utrzymanie serwerów o wysokiej przepustowości oraz niezawodności, co prowadzi do koncentracji władzy w rękach dużych organizacji dysponujących odpowiednimi zasobami.

W przeciwieństwie do architektury klient-serwer, model peer-to-peer zakłada równorzędność wszystkich węzłów uczestniczących w sieci. Każdy uczestnik może jednocześnie pełnić rolę zarówno dostawcy, jak i konsumenta zasobów, co prowadzi do symetrycznego rozkładu obciążenia oraz eliminacji konieczności utrzymywania dedykowanych serwerów centralnych. Taka architektura charakteryzuje się naturalną skalowalnością, gdyż każdy nowy węzeł dołączający do sieci jednocześnie zwiększa jej zasoby obliczeniowe oraz przepustowość. Ponadto, systemy P2P wykazują wysoką odporność na awarie poje-

dynczych węzłów, gdyż utrata części uczestników nie prowadzi do utraty funkcjonalności całego systemu.

Systemy peer-to-peer można sklasyfikować według różnych kryteriów, przy czym najistotniejszym z nich jest stopień strukturyzacji sieci. Sieci nieustrukturyzowane charakteryzują się brakiem z góry określonej topologii oraz przypadkowym rozkładem połączeń pomiędzy węzłami. Wyszukiwanie zasobów w takich sieciach opiera się zazwyczaj na mechanizmach rozgłaszania zapytań do sąsiadujących węzłów[10], co może prowadzić do wysokiego obciążenia sieci, lecz jednocześnie zapewnia dużą odporność na zmiany topologii. Przykładami takich rozwiązań są wczesne systemy wymiany plików, takie jak Gnutella.

Sieci strukturyzowane wprowadzają natomiast uporządkowaną topologię, w której położenie każdego węzła oraz rozmieszczenie zasobów podlega określonym regułom matematycznym. Najpopularniejszą strukturą tego typu są rozproszone tablice haszujące, w których każdy węzeł odpowiada za określony fragment przestrzeni kluczy, a routing zapytań odbywa się zgodnie z algorytmami zapewniającymi logarytmiczną złożoność czasową wyszukiwania. Rozwiązania takie, omówione szczegółowo w dalszej części niniejszego rozdziału, oferują gwarancje dotyczące czasu odnalezienia zasobu oraz bardziej efektywne wykorzystanie zasobów sieciowych w porównaniu do sieci nieustrukturyzowanych.

Pośrednim rozwiązaniem są sieci hybrydowe, które łączą elementy decentralizacji z wybranymi komponentami centralizacji. Przykładowo, mogą one wykorzystywać specjalne węzły o podwyższonych uprawnieniach, odpowiedzialne za indeksowanie zasobów lub koordynację działań, przy jednoczesnym zachowaniu peer-to-peer dla bezpośredniej wymiany danych między użytkownikami. Takie podejście pozwala na osiągnięcie kompromisu pomiędzy efektywnością wyszukiwania a stopniem decentralizacji systemu.

Realizacja systemów peer-to-peer wiąże się z szeregiem wyzwań technicznych, które nie występują lub mają mniejsze znaczenie w architekturze klient-serwer. Jednym z

[10]Rozgłaszanie (ang. *flooding*) to technika rozprzestrzeniania informacji w sieci, polegająca na przekazywaniu komunikatu do wszystkich sąsiadujących węzłów, które następnie powtarzają ten proces, aż do osiągnięcia określonego limitu kroków.

kluczowych problemów jest mechanizm wykrywania innych uczestników sieci, szczególnie w sytuacji, gdy węzły dynamicznie dołączają oraz opuszczają sieć. W sieciach lokalnych problem ten można rozwiązać przy użyciu protokołów rozgłoszeniowych, takich jak Multicast DNS, natomiast w sieciach rozległych konieczne jest zastosowanie rozproszonych mechanizmów indeksowania oraz routingu.

Kolejnym istotnym wyzwaniem jest przechodzenie przez translację adresów sieciowych, która stanowi istotną barierę dla bezpośredniej komunikacji P2P w środowisku internetu. Większość urządzeń końcowych znajduje się w sieciach prywatnych za routerami wykonującymi translację adresów, co uniemożliwia nawiązywanie połączeń przychodzących bez dodatkowych mechanizmów, takich jak STUN czy TURN. Rozwiązanie tego problemu wymaga zastosowania zaawansowanych technik traversal lub rezygnacji z pełnej decentralizacji na rzecz węzłów pośredniczących.

Wreszcie, kluczowym zagadnieniem w systemach P2P jest zapewnienie spójności danych oraz odporności na manipulacje ze strony złośliwych uczestników. W środowisku pozbawionym zaufanej instancji centralnej konieczne jest zastosowanie mechanizmów kryptograficznych oraz algorytmów konsensusu[11], które pozwalają na weryfikację autentyczności danych oraz wykrywanie prób oszustw, nawet w obecności znaczącej liczby niewspółpracujących węzłów.

Pomimo wymienionych wyzwań, systemy peer-to-peer znajdują szerokie zastosowanie w różnorodnych domenach aplikacyjnych. Oprócz pierwotnego wykorzystania w systemach wymiany plików, architektura P2P stanowi fundament współczesnych systemów rozproszonej pamięci masowej, protokołów komunikacyjnych odpornych na cenzurę, systemów blockchain oraz rozproszonych platform obliczeniowych. Niniejszy projekt wpisuje się w nurt zastosowań komunikacyjnych, oferując zdecentralizowaną alternatywę dla tradycyjnych komunikatorów internetowych.

[11]Algorytmy konsensusu (ang. *consensus algorithms*) to protokoły umożliwiające grupie rozproszonych węzłów osiągnięcie zgodności co do stanu systemu pomimo obecności awarii lub złośliwych uczestników. Przykładami są algorytmy typu Paxos, Raft czy protokoły wykorzystywane w systemach blockchain.

C1.1.2. Podstawy kryptografii krzywych eliptycznych

Kryptografia krzywych eliptycznych (ECC) jest dziś jednym z podstawowych narzędzi kryptografii klucza publicznego. Umożliwia uzyskanie poziomu bezpieczeństwa porównywalnego z klasycznymi schematami, takimi jak RSA, przy znacznie krótszych kluczach, co przekłada się na mniejszy narzut obliczeniowy i sieciowy[12].

W projekcie wykorzystano krzywą Curve25519 oraz funkcję X25519, która realizuje wymianę kluczy Diffie-Hellmana na tej krzywej. X25519 została zaprojektowana z myślą o bezpiecznej implementacji (m.in. odporności na typowe ataki czasowe) i jest szeroko dostępna w dojrzałych bibliotekach kryptograficznych.

Wykorzystanie krzywej Curve25519 w kontekście niniejszego projektu wynika z połączenia wysokiego poziomu bezpieczeństwa, efektywności obliczeniowej oraz dostępności dojrzałych implementacji bibliotecznych. Funkcja X25519 służy do uzgadniania wspólnego sekretu pomiędzy komunikującymi się stronami, na podstawie którego następnie derywowany jest klucz symetryczny wykorzystywany do szyfrowania treści wiadomości. Taka konstrukcja łączy zalety kryptografii asymetrycznej, umożliwiającej bezpieczną wymianę kluczy bez wcześniejszego ustanawiania współdzielonego sekretu, z wydajnością algorytmów symetrycznych przy szyfrowaniu dużych ilości danych.

C1.1.3. Szyfrowanie uwierzytelnione z danymi dodatkowymi

Szyfrowanie uwierzytelnione z danymi dodatkowymi (AEAD) łączy w jednym prymitywie szyfrowanie i weryfikację integralności danych. Zamiast osobno stosować szyfr oraz kod uwierzytelniający, algorytm AEAD przyjmuje klucz, nonce[13] oraz dane (i ewentualne dane dodatkowe), a zwraca szyfrogram wraz z krótkim tagiem[14] pozwalającym wykryć jakąkolwiek modyfikację.

[12] Szerokie omówienie podstaw kryptografii klucza publicznego oraz schematów opartych na krzywych eliptycznych zawierają m.in. Menezes i in., *Handbook of Applied Cryptography* [32] oraz Katz, Lindell, *Introduction to Modern Cryptography* [33].

[13] Nonce (skrót od *number used once*) to wartość używana jednorazowo w danym kontekście kryptograficznym; zapewnia, że szyfrowanie tej samej wiadomości tym samym kluczem przy różnych nonce prowadzi do odmiennych szyfrogramów.

[14] Tag uwierzytelniający (ang. *authentication tag*) to krótka wartość dołączona do szyfrogramu, umożliwiająca odbiorcy wykrycie jakiejkolwiek modyfikacji zaszyfrowanych danych lub danych dodatkowych.

W projekcie zastosowano algorytm ChaCha20-Poly1305, standardowy schemat AEAD opisany w RFC 8439 (Nir, Langley, *ChaCha20 and Poly1305 for IETF Protocols*). ChaCha20 generuje strumień pseudolosowych bajtów, które są łączone z tekstem jawnym, a Poly1305 wylicza tag uwierzytelniający. Klucz symetryczny derywowany jest ze wspólnego sekretu ustalonego przy użyciu X25519, natomiast nonce generowany jest losowo dla każdej wiadomości, co zapewnia różne szyfrogramy nawet dla identycznych treści i uniemożliwia niezauważoną manipulację.

C1.1.4. Protokół Noise Framework

Ustanowienie bezpiecznego kanału komunikacyjnego w systemie peer-to-peer wymaga protokołu, który bez udziału centralnego serwera pozwala uzgodnić klucze sesyjne i wzajemnie uwierzytelnić strony. W niniejszym projekcie rolę tę pełni Noise Protocol Framework[15], wykorzystywany jako warstwa bezpieczeństwa transportu w bibliotece `libp2p`.

Noise definiuje schematy wymiany komunikatów (tzw. wzorce handshake), w których strony wymieniają swoje klucze publiczne, wyznaczają wspólne sekrety i derywują z nich klucze sesyjne. Kluczową cechą jest użycie efemerycznych kluczy[16] dla każdej sesji, co zapewnia poufność postępującą[17] – przechwycenie długoterminowych kluczy tożsamości nie pozwala na odszyfrowanie wcześniej nagranego ruchu.

W kontekście biblioteki `libp2p`, wykorzystywanej w niniejszym projekcie, protokół Noise służy do zabezpieczania połączeń TCP pomiędzy węzłami sieci. Każde połączenie pomiędzy peerami rozpoczyna się od handshake Noise, po którym cała dalsza komunikacja jest szyfrowana i uwierzytelniana na poziomie transportu, niezależnie od dodatkowych mechanizmów szyfrowania zastosowanych w warstwie aplikacyjnej.

Istotną zaletą Noise w porównaniu z TLS jest prostota implementacji oraz weryfikacji poprawności. Specyfikacja protokołu jest zwięzła oraz precyzyjna, co ułatwia

[15] Specyfikację frameworka Noise przedstawia T. Perrin, *The Noise Protocol Framework*, omówioną szerzej w dokumentacji projektu Noise oraz bibliografii niniejszej pracy [10].

[16] Klucz efemeryczny (ang. *ephemeral key*) to klucz kryptograficzny generowany wyłącznie na potrzeby jednej sesji komunikacyjnej i usuwany po jej zakończeniu, co ogranicza skutki ewentualnego wycieku.

[17] Poufność postępująca (ang. *forward secrecy*) to właściwość protokołu kryptograficznego, w której przejęcie długoterminowych kluczy nie umożliwia odszyfrowania wcześniej przechwyconej komunikacji, ponieważ wykorzystywała ona niezależne, efemeryczne klucze sesyjne.

formalne dowodzenie właściwości bezpieczeństwa oraz redukuje ryzyko błędów implementacyjnych. Ponadto, Noise nie wymaga złożonej infrastruktury certyfikatów ani negocjacji parametrów kryptograficznych, co upraszcza wdrożenie oraz redukuje powierzchnię ataku. Dla systemów P2P, gdzie nie istnieje naturalna hierarchia zaufania, takie właściwości są szczególnie wartościowe.

C1.1.5. Rozproszone tablice haszujące

Rozproszone tablice haszujące (DHT) rozwiązują problem wyszukiwania danych w dużych, zdecentralizowanych sieciach, w których nie istnieje centralny serwer indeksujący. Zarówno węzłom, jak i kluczom zasobów przypisuje się identyfikatory z tej samej przestrzeni adresowej, a każdy węzeł odpowiada za przechowywanie danych o kluczach „bliskich" jego identyfikatorowi zgodnie z przyjętą metryką.

W projekcie wykorzystano DHT opartą na algorytmie Kademlia, dostarczaną przez bibliotekę `libp2p-kad`. Kademlia definiuje odległość między identyfikatorami przy użyciu metryki XOR, co pozwala na efektywny routing zapytań: każde zapytanie o dany klucz jest kierowane iteracyjnie do węzłów coraz bliższych temu kluczowi, a liczba wymaganych kroków rośnie logarytmicznie wraz z rozmiarem sieci.

Biblioteka `libp2p-kad` utrzymuje wewnętrznie strukturę *k-buckets* przechowującą informacje o znanych węzłach w różnych zakresach odległości, zapewniając kompromis między zużyciem pamięci a szybkością wyszukiwania. Replikacja danych na wielu węzłach bliskich danemu kluczowi zwiększa dostępność zasobów mimo zmian topologii i awarii części uczestników.

W kontekście niniejszego projektu DHT Kademlia wykorzystywana jest przede wszystkim do odkrywania węzłów pełniących rolę dostawców usługi mailbox. Węzły deklarujące gotowość do przechowywania wiadomości dla nieobecnych użytkowników publikują odpowiednie rekordy w DHT, a klienci odnajdują dostawców poprzez zapytania `get_providers`. Pozwala to realizować rozproszony rejestr usług mailbox bez centralnego serwera, zgodnie z rzeczywistą implementacją w warstwie `libp2p`.

C1.1.6. Multicast DNS

Wykrywanie usług oraz rozwiązywanie nazw w sieciach lokalnych tradycyjnie wymagało konfiguracji centralnego serwera DNS lub ręcznego wprowadzania adresów IP przez użytkowników. Takie podejście, choć funkcjonalne w środowiskach korporacyjnych z dedykowaną administracją sieciową, stanowi istotną barierę dla użyteczności w sieciach domowych oraz innych środowiskach pozbawionych wyspecjalizowanej infrastruktury. Multicast DNS stanowi protokół zaprojektowany w celu eliminacji tej bariery poprzez umożliwienie automatycznego wykrywania oraz rozwiązywania nazw bez konieczności konfiguracji.

Protokół mDNS, zdefiniowany w dokumencie RFC 6762, wykorzystuje mechanizm multiemisji IP[18] do rozpowszechniania zapytań oraz odpowiedzi DNS w obrębie sieci lokalnej. Zamiast wysyłać zapytania do dedykowanego serwera, węzły rozgłaszają je na specjalny adres multiemisji, a urządzenia oferujące daną usługę odpowiadają bezpośrednio w tej samej grupie.

W praktyce pozwala to na automatyczne wykrywanie usług w sieci lokalnej bez konieczności ręcznej konfiguracji adresów IP czy serwera DNS. W niniejszym projekcie komponent `libp2p-mdns` wykorzystuje mDNS do wykrywania innych instancji aplikacji uruchomionych w tej samej podsieci. Przy starcie węzeł publikuje swoją obecność, a zdarzenia `Discovered` oraz `Expired` są obsługiwane w module `network/handlers/discovery.rs`, gdzie adresy wykrytych peerów są dodawane do tablicy routingu i, jeśli to możliwe, następuje próba nawiązania połączenia.

mDNS działa wyłącznie w obrębie pojedynczej sieci lokalnej i nie przechodzi przez routery, co z jednej strony ogranicza zakres zastosowań do środowisk LAN, a z drugiej strony zmniejsza powierzchnię ataku – informacje o usługach nie są domyślnie widoczne poza lokalną podsiecią.

[18]Multiemisja IP (ang. *IP multicast*) to metoda przesyłania pakietów sieciowych do grupy odbiorców jednocześnie, w przeciwieństwie do unicast (jeden odbiorca) oraz broadcast (wszyscy odbiorcy w sieci). Pakiety multiemisji są dostarczane wyłącznie do węzłów, które wyraziły zainteresowanie odbiorem danych z danej grupy multiemisji.

C1.1.7. Synteza założeń teoretycznych

Przedstawione fundamenty teoretyczne tworzą spójną podstawę dla zaprojektowanego systemu zdecentralizowanej komunikacji. Architektura peer-to-peer eliminuje pojedyncze punkty awaryjne oraz centralizację kontroli, kryptografia krzywych eliptycznych i szyfrowanie uwierzytelnione zapewniają poufność oraz integralność danych, a protokół Noise odpowiada za bezpieczne kanały transportowe. Mechanizmy DHT i mDNS rozwiązują z kolei problem wykrywania oraz lokalizacji węzłów w środowisku rozproszonym, umożliwiając pełne odejście od centralnych serwerów.

Kluczowym wyzwaniem jest takie połączenie tych komponentów, aby utrzymać rozsądny kompromis pomiędzy bezpieczeństwem, wydajnością a użytecznością. Zbyt rozbudowane mechanizmy ochronne mogłyby obniżyć komfort użytkowania i wydajność, natomiast nadmierne uproszczenia prowadziłyby do podatności. W dalszej części pracy pokazano, w jaki sposób opisane koncepcje zostały przełożone na konkretną architekturę i implementację komunikatora P2P działającego w sieci lokalnej.

C1.2. Opis sytuacji faktycznej

Zrealizowany system stanowi w pełni funkcjonalny zdecentralizowany komunikator peer-to-peer działający w sieci lokalnej, zaimplementowany w języku Rust. Aplikacja umożliwia uruchomienie dowolnej liczby węzłów klienckich oraz węzłów typu mailbox, które automatycznie odnajdują się nawzajem w obrębie tej samej podsieci lokalnej. System zapewnia pełne szyfrowanie end-to-end, asynchroniczną wymianę wiadomości poprzez mechanizm przechowywania dla użytkowników tymczasowo nieobecnych w sieci oraz intuicyjny interfejs użytkownika zarówno w postaci terminalowej, jak i przeglądarkowej opartej na frameworku Vue.js.

C1.2.1. Architektura systemu oraz aktorzy

W systemie wyróżnić można trzy główne typy aktorów, których interakcje tworzą funkcjonalną całość komunikatora. Użytkownik końcowy stanowi podstawowego beneficjenta systemu, inicjując połączenia z siecią, wysyłając oraz odbierając zaszyfrowane wiadomości tekstowe, zarządzając listą kontaktów oraz przeglądając historię konwersacji. Użytkownik wchodzi w interakcję z systemem poprzez dwa dostępne interfejsy: termina-

lowy interfejs tekstowy przeznaczony dla zaawansowanych użytkowników oraz administratorów, a także responsywny interfejs przeglądarkowy zapewniający intuicyjną obsługę dla przeciętnych użytkowników. Węzeł kliencki automatycznie wykrywa obecność innych uczestników w sieci lokalnej, zarządza procesami kryptograficznymi oraz odpowiada za lokalne przechowywanie historii komunikacji w sposób zapewniający poufność danych nawet w przypadku fizycznego dostępu do urządzenia.

Drugim kluczowym aktorem są węzły typu mailbox, które pełnią rolę rozproszonego systemu przechowywania wiadomości przeznaczonych dla tymczasowo nieobecnych użytkowników. Węzły te deklarują swoją gotowość do świadczenia usługi poprzez publikację odpowiednich rekordów w rozproszonej tablicy haszującej, co umożliwia innym uczestnikom sieci odnalezienie ich w sposób zdecentralizowany. Mailbox przyjmuje zaszyfrowane wiadomości od nadawców, przechowuje je w sposób zapewniający poufność oraz integralność, a następnie udostępnia odbiorcom po ich powrocie do sieci. Istotną cechą tej architektury jest fakt, że węzeł mailbox nie posiada możliwości odczytania treści przechowywanych wiadomości, gdyż są one szyfrowane kluczem znanym wyłącznie nadawcy oraz odbiorcy, zgodnie z modelem E2EE stosowanym również w nowoczesnych komunikatorach takich jak Signal (por. Marlinspike, Perrin, *The Double Ratchet Algorithm*).

Trzecim aktorem jest infrastruktura sieciowa peer-to-peer, która obejmuje mechanizmy wykrywania węzłów, routingu zapytań oraz utrzymywania rozproszonej tablicy haszującej. Infrastruktura ta działa w sposób całkowicie autonomiczny, nie wymagając interwencji użytkownika ani obecności centralnych serwerów koordynujących. Protokół mDNS, zdefiniowany w RFC 6762 (Cheshire, Krochmal, *Multicast DNS*), umożliwia automatyczne wykrywanie węzłów w sieci lokalnej poprzez mechanizm multiemisji, natomiast DHT oparta na algorytmie Kademlia (Maymounkov, Mazières, *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*) zapewnia efektywne odnajdywanie węzłów mailbox w szerszym kontekście sieciowym.

C1.2.2. Przypadki użycia oraz scenariusze interakcji

System został zaprojektowany w celu zapewnienia użytkownikom intuicyjnej obsługi przy jednoczesnym zachowaniu wysokiego poziomu bezpieczeństwa. Podstawowym scenariuszem jest automatyczne wykrywanie nowych uczestników sieci. Po uruchomieniu aplikacji każdy węzeł rozgłasza w sieci lokalnej komunikaty ogłoszeniowe wykorzystujące protokół mDNS, zawierające podstawowe informacje identyfikacyjne oraz klucze publiczne. Pozostałe aktywne węzły odbierają te komunikaty, weryfikują ich autentyczność oraz automatycznie dodają nowego uczestnika do lokalnej książki adresowej, eliminując konieczność ręcznej konfiguracji połączeń przez użytkowników.

Komunikacja bezpośrednia pomiędzy użytkownikami przebiega w sposób w pełni zaszyfrowany. Po wybraniu rozmówcy z listy dostępnych kontaktów, system automatycznie uzgadnia klucz symetryczny wykorzystując wcześniej wymienione klucze publiczne algorytmu X25519. Treść wiadomości jest następnie szyfrowana przy użyciu algorytmu ChaCha20-Poly1305, który zapewnia zarówno poufność, jak i integralność przesyłanych danych. Zaszyfrowana wiadomość przesyłana jest poprzez protokół request-response biblioteki libp2p, który zapewnia niezawodne dostarczenie oraz potwierdzenie odbioru. Cała warstwa transportu jest dodatkowo zabezpieczona protokołem Noise, co tworzy wielowarstwową ochronę przed potencjalnymi atakami.

Odbiór wiadomości odbywa się w sposób automatyczny oraz transparentny dla użytkownika. Każda przychodząca wiadomość zostaje odszyfrowana przy użyciu klucza wspólnego ustalonego z nadawcą, po czym weryfikowana jest jej integralność poprzez sprawdzenie tagu uwierzytelniającego. Po pomyślnej weryfikacji wiadomość zapisywana jest w lokalnej bazie danych sled wraz z metadanymi obejmującymi identyfikator nadawcy, znacznik czasowy oraz kierunek transmisji. W bazie przechowywana jest pełna treść wiadomości oraz powiązane metadane, a użytkownik może opcjonalnie włączyć dodatkowe szyfrowanie danych w spoczynku przy użyciu hasła głównego.

Przeglądanie historii konwersacji umożliwia użytkownikom dostęp do pełnego archiwum wymienianych wiadomości. System oferuje możliwość filtrowania rozmów

według rozmówcy oraz stronicowania[19] dla wydajnego wyświetlania długich historii. Wszystkie operacje na historii zostały zoptymalizowane poprzez wykorzystanie kluczy kompozytowych w bazie danych, umożliwiających efektywne zapytania zakresowe oraz sortowanie chronologiczne.

Kluczową funkcjonalnością zapewniającą asynchroniczną komunikację jest mechanizm przechowywania wiadomości dla użytkowników tymczasowo nieobecnych. W sytuacji gdy odbiorca nie jest dostępny w sieci, nadawca automatycznie lokalizuje dostępne węzły mailbox poprzez zapytania do DHT. Wiadomość jest następnie szyfrowana kluczem publicznym odbiorcy oraz przesyłana do wybranych węzłów mailbox w celu przechowywania. Dla zapewnienia niezawodności system replikuje każdą wiadomość na co najmniej dwóch niezależnych węzłach mailbox. Po powrocie odbiorcy do sieci jego klient automatycznie odpytuje znane węzły mailbox o oczekujące wiadomości, pobiera je, deszyfruje oraz potwierdza odbiór, prowadząc do usunięcia wiadomości z węzłów przechowujących.

Zarządzanie cyklem życia połączeń oraz kluczy kryptograficznych odbywa się w sposób automatyczny. Warstwa sieciowa wykorzystuje wbudowany w libp2p mechanizm ping do okresowego sprawdzania dostępności peerów oraz reaguje na zdarzenia nawiązywania i zrywania połączeń pochodzące z warstwy `Swarm`. W przypadku wykrycia trwałej niedostępności uczestnika odpowiadające mu dane kryptograficzne mogą zostać usunięte w celu zwolnienia zasobów, przy czym historia konwersacji jest zachowywana dla zapewnienia ciągłości komunikacji po ewentualnym powrocie uczestnika.

C1.2.3. Przebieg typowej sesji komunikacyjnej

Typowa sesja użytkownika rozpoczyna się od uruchomienia aplikacji, podczas którego system inicjalizuje wszystkie niezbędne komponenty, w tym ładuje lub generuje pary kluczy kryptograficznych, otwiera lokalną bazę danych oraz uruchamia stos sieciowy libp2p. Użytkownik może wybrać pomiędzy interfejsem terminalowym a interfejsem prze-

[19]Stronicowanie (ang. *pagination*) to technika dzielenia dużych zbiorów danych na mniejsze strony wyświetlane sekwencyjnie, zapewniająca efektywne przeglądanie obszernych kolekcji bez konieczności jednoczesnego ładowania wszystkich elementów.

glądarkowym, przy czym w przypadku wyboru interfejsu przeglądarkowego uruchamiany jest lokalny serwer HTTP obsługujący komunikację poprzez protokół WebSocket dla zapewnienia aktualizacji stanu w czasie rzeczywistym.

Natychmiast po uruchomieniu węzeł rozpoczyna proces wykrywania innych uczestników poprzez emisję komunikatów mDNS w sieci lokalnej. Jednocześnie nasłuchuje on komunikatów od innych węzłów, budując dynamiczną książkę adresową dostępnych peerów. Dla każdego wykrytego peera system automatycznie wymienia klucze publiczne, które są następnie zapisywane w lokalnej bazie danych wraz z opcjonalnymi pseudonimami ułatwiającymi identyfikację przez użytkownika.

Wysłanie pierwszej wiadomości do wybranego kontaktu inicjuje proces uzgadniania wspólnego sekretu poprzez wymianę kluczy Diffie-Hellmana na krzywej eliptycznej. Z ustalonego wspólnego sekretu derywowany jest klucz symetryczny, który następnie wykorzystywany jest do szyfrowania treści wszystkich kolejnych wiadomości wymienianych z tym rozmówcą. Po zaszyfrowaniu wiadomość przesyłana jest przez dedykowany kanał komunikacyjny ustanowiony między peerami, a system oczekuje na potwierdzenie dostarczenia.

Odbiorca, po otrzymaniu wiadomości, automatycznie deszyfruje jej zawartość wykorzystując odpowiedni klucz wspólny, weryfikuje integralność oraz zapisuje w lokalnej historii. Użytkownik jest natychmiast powiadamiany o nowej wiadomości poprzez odpowiednie zdarzenie w interfejsie użytkownika, przy czym w przypadku interfejsu przeglądarkowego aktualizacja następuje w czasie rzeczywistym dzięki wykorzystaniu protokołu WebSocket. Historia wszystkich wymienianych wiadomości jest przechowywana lokalnie w sposób zapewniający ich dostępność nawet po restarcie aplikacji lub okresowej niedostępności rozmówców.

Zrealizowany system zapewnia pełną funkcjonalność komunikatora peer-to-peer, obejmującą bezpieczną wymianę wiadomości, asynchroniczne dostarczanie poprzez mechanizm mailbox, responsywny interfejs przeglądarkowy oraz pełną kontrolę użytkownika nad własnymi danymi. Modularna architektura systemu umożliwia potencjalną rozbudowę o dodatkowe funkcjonalności, takie jak przesyłanie plików, rozmowy głosowe lub gru-

powe czaty, przy jednoczesnym zachowaniu fundamentalnych zasad bezpieczeństwa oraz decentralizacji.

C1.3. Badania własne / opis metod, technik i narzędzi badawczych / aparatura / oprogramowanie

Metodologia pracy nad projektem i metody badawcze

Projekt rozwijano w podejściu iteracyjnym. Nowe funkcjonalności były wprowadzane w małych krokach, a po większych zmianach wykonywano testy manualne. Poszczególne moduły powstawały na podstawie analizy wymagań funkcjonalnych oraz dokumentacji technicznej bibliotek wykorzystywanych w projekcie. Działanie systemu weryfikowano poprzez uruchamianie kilku instancji programu w sieci lokalnej i testowanie scenariuszy typowych dla komunikatora: wykrywania peerów z wykorzystaniem mDNS, wysyłania i odbierania zaszyfrowanych wiadomości, działania mechanizmu mailbox przy nieobecnym odbiorcy oraz zachowania aplikacji przy kilku równoległych konwersacjach.

Wybór języka programowania Rust i jego przewagi

Jako język implementacji wybrano Rust, nowoczesny język systemowy łączący wydajność zbliżoną do C/C++ z rozbudowanymi mechanizmami bezpieczeństwa pamięci. System ownership[20] oraz weryfikacja pożyczania referencji przez borrow checker[21] pozwalają wykryć wiele błędów na etapie kompilacji. Brak garbage collectora[22] umożliwia przewidywalne zużycie zasobów, co jest istotne w aplikacjach sieciowych działających długotrwale.

Dzięki tym mechanizmom ryzyko typowych błędów związanych z zarządzaniem

[20] Ownership (ang. *własność*) to system zarządzania pamięcią w języku Rust, w którym każda wartość ma dokładnie jednego właściciela, a pamięć jest automatycznie zwalniana, gdy właściciel wychodzi poza zakres. Eliminuje to wycieki pamięci bez konieczności garbage collectora.

[21] Borrow checker to mechanizm kompilatora Rust weryfikujący w czasie kompilacji poprawność pożyczania referencji do danych, ograniczający możliwość powstawania jednoczesnych modyfikacji tych samech danych i wielu warunków wyścigu.

[22] Garbage collector (GC) to mechanizm automatycznego zarządzania pamięcią, który okresowo skanuje pamięć w poszukiwaniu nieużywanych obiektów. Rust osiąga automatyczne zarządzanie pamięcią bez GC poprzez system ownership, eliminując pauzy charakterystyczne dla języków z GC.

pamięcią (takich jak use-after-free[23], double free[24] czy przepełnienia bufora[25]) w kodzie napisanym w bezpiecznym podzbiorze języka jest znacząco zredukowane, choć nie eliminuje to całkowicie możliwości wystąpienia podatności. Rust jest szeroko stosowany w projektach związanych z kryptografią, blockchainem oraz programowaniem sieciowym, a bogaty ekosystem bibliotek (*crates.io*) oraz narzędzie Cargo ułatwiają zarządzanie zależnościami i budowaniem projektu.

Biblioteka `libp2p` jako fundament komunikacji P2P

Sercem projektu jest biblioteka `libp2p`, zaprojektowana do obsługi komunikacji peer-to-peer w duchu decentralizacji i modularności. Biblioteka libp2p umożliwia komponowanie własnego stosu sieciowego poprzez dobór protokołów transportowych, systemów wymiany wiadomości oraz mechanizmów odkrywania peerów.

W projekcie wykorzystano szereg komponentów ekosystemu libp2p, wśród których kluczową rolę odgrywa **libp2p-mdns** zapewniający automatyczne wykrywanie peerów w sieci lokalnej poprzez Multicast DNS, eliminując konieczność ręcznej konfiguracji adresów IP czy portów. Protokół **libp2p-request-response** implementuje bezstanową semantykę żądanie–odpowiedź, odpowiednią dla architektury komunikatora. Warstwa bezpieczeństwa transportu realizowana jest przez **libp2p-noise**, który zestawia bezpieczne połączenia według protokołu Noise opartego na krzywych eliptycznych, zapewniając poufność transmisji na poziomie transportu. Do multipleksowania[26] równoczesnych strumieni danych wykorzystano **libp2p-yamux**, umożliwiający prowadzenie wielu niezależ-

[23]Use-after-free to podatność polegająca na dostępie do pamięci po jej zwolnieniu, mogąca prowadzić do wykonania arbitralnego kodu.

[24]Double free to błąd polegający na dwukrotnym zwolnieniu tej samej pamięci, prowadzący do korupcji struktur zarządzania pamięcią.

[25]Buffer overflow (ang. *przepełnienie bufora*) to podatność polegająca na zapisie danych poza granicami zaalokowanego bufora, umożliwiająca nadpisanie sąsiedniej pamięci oraz potencjalne wykonanie arbitralnego kodu.

[26]Multipleksowanie (ang. *multiplexing*) to technika przesyłania wielu niezależnych strumieni danych przez jedno fizyczne połączenie sieciowe, umożliwiająca efektywne wykorzystanie zasobów oraz prowadzenie równoczesnych konwersacji bez konieczności nawiązywania osobnych połączeń TCP dla każdego strumienia.

nych konwersacji przez jedno połączenie TCP. Obsługa rozproszonych tablic haszujących zrealizowana jest przez **libp2p-kad**, implementujący algorytm Kademlia DHT wykorzystywany w projekcie do wykrywania węzłów mailbox oraz ogłaszania dostępności usług przechowywania wiadomości dla użytkowników tymczasowo nieobecnych.

Techniki i narzędzia programistyczne

Jako główny język implementacji wybrano Rust ze względu na połączenie wysokiej wydajności z mechanizmami bezpieczeństwa pamięci (system własności, sprawdzanie pożyczania, brak garbage collectora) oraz dostępność bibliotek kryptograficznych i sieciowych w repozytorium `crates.io`. Rust zapewnia także dobre wsparcie dla programowania asynchronicznego, co jest istotne w aplikacjach sieciowych.

Do przechowywania danych lokalnych wykorzystano bazę danych **sled**, będącą bazą typu klucz–wartość napisaną w języku Rust. W projekcie służy ona do przechowywania historii konwersacji, listy kontaktów, kolejek wiadomości wychodzących oraz wiadomości oczekujących w skrzynce mailbox. Wykorzystanie drzew klucz–wartość oraz uporządkowanych kluczy pozwala na efektywne przeszukiwanie danych bez konieczności użycia relacyjnego silnika bazodanowego.

Warstwa kryptograficzna opiera się na bibliotekach:
`x25519-dalek`, `chacha20poly1305` oraz `sha2`.
Biblioteka `x25519-dalek` jest wykorzystywana do uzgadniania kluczy metodą Diffie-Hellmana na krzywej Curve25519, `chacha20poly1305` do uwierzytelnionego szyfrowania AEAD, a `sha2` do derywacji kluczy i obliczania sum kontrolnych. Zastosowane algorytmy są zgodne z opisem w RFC 7748 oraz RFC 8439 i są powszechnie używane w nowoczesnych protokołach internetowych.

Programowanie asynchroniczne realizowane jest przy użyciu runtime'u[27] `tokio`, który zapewnia obsługę operacji wejścia-wyjścia bez blokowania wątków, oraz biblioteki `futures` dostarczającej abstrakcje dla operacji asynchronicznych. Tokio umożliwia jed-

[27]Runtime (ang. *środowisko uruchomieniowe*) to warstwa oprogramowania zapewniająca usługi niezbędne do wykonywania programów, takie jak zarządzanie pamięcią, operacje wejścia-wyjścia czy scheduler zadań. W kontekście Tokio, runtime zarządza pętlą zdarzeń oraz szeregowaniem zadań asynchronicznych.

noczesną obsługę wielu zadań, w tym równoczesnych połączeń sieciowych, komunikacji z użytkownikiem poprzez interfejs oraz operacji na bazie danych.

Interfejs przeglądarkowy zrealizowano przy użyciu frameworka **Axum**, pełniącego rolę serwera HTTP w części serwerowej, oraz frameworka **Vue.js 3** po stronie przeglądarki. Axum obsługuje zarówno punkty dostępu REST API[28], jak i komunikację w czasie rzeczywistym poprzez WebSocket[29]. Interfejs użytkownika po stronie przeglądarki zaimplementowano w Vue.js 3 z wykorzystaniem Composition API[30] oraz zarządzania stanem aplikacji poprzez bibliotekę **Pinia**. Statyczne pliki interfejsu są osadzane w binarce aplikacji za pomocą biblioteki `rust_embed`, co upraszcza dystrybucję systemu.

Interfejs terminalowy (TUI) wykorzystuje bibliotekę `crossterm` do obsługi wejścia z klawiatury i rysowania w terminalu. Argumenty wiersza poleceń przy uruchamianiu programu są przetwarzane za pomocą biblioteki `clap`. Parsowanie poleceń wpisywanych w TUI realizowane jest przez wewnętrzny mechanizm oparty na podziale wejścia na tokeny. Dodatkowe biblioteki pomocnicze obejmują `base64` oraz `hex` do kodowania danych binarnych oraz `tracing` zapewniającą logowanie strukturalne.

Środowisko testowe obejmowało uruchamianie wielu instancji aplikacji w sieci lokalnej, z wykorzystaniem zarówno połączeń przewodowych, jak i bezprzewodowych. Do analizy ruchu sieciowego wykorzystywano narzędzie Wireshark[31], w szczególności w celu potwierdzenia, że treść wiadomości nie jest przesyłana w postaci niezaszyfrowanej.

Architektura modułowa aplikacji

[28] REST API (Representational State Transfer Application Programming Interface) to architektura interfejsów programistycznych oparta na protokole HTTP, wykorzystująca metody HTTP (GET, POST, PUT, DELETE) do manipulacji zasobami reprezentowanymi zazwyczaj w formacie JSON lub XML.

[29] WebSocket to protokół komunikacyjny zapewniający pełnodupleksowe kanały komunikacyjne przez pojedyncze połączenie TCP, umożliwiający dwukierunkową wymianę wiadomości między klientem a serwerem w czasie rzeczywistym, bez konieczności ciągłego odpytywania serwera.

[30] Composition API to sposób organizacji komponentów w frameworku Vue.js 3, wykorzystujący funkcje zamiast obiektów opcji, co ułatwia kompozycję logiki i ponowne wykorzystanie kodu.

[31] Wireshark - otwartoźródłowe narzędzie do przechwytywania oraz analizy pakietów sieciowych, umożliwiające inspekcję protokołów komunikacyjnych na różnych warstwach modelu OSI. Dostępne na: *https://www.wireshark.org*

System zorganizowany jest w postaci modularnej architektury, w której każdy moduł odpowiada za wydzielony obszar funkcjonalności, co ułatwia utrzymanie oraz rozwijanie kodu. Główne moduły systemu obejmują warstwę kryptograficzną, warstwę sieciową, warstwę persystencji danych, silnik synchronizacji oraz interfejsy użytkownika.

Moduł kryptograficzny (crypto/) odpowiada za operacje związane z szyfrowaniem, deszyfrowaniem oraz zarządzaniem kluczami. Moduł `crypto/identity.rs` zarządza tożsamością użytkownika, obejmującą parę kluczy Ed25519[32] wykorzystywaną przez libp2p do uwierzytelniania węzła w sieci P2P oraz parę kluczy X25519 służącą do uzgadniania wspólnych sekretów metodą Diffie-Hellmana. Plik `crypto/hpke.rs` implementuje uproszczony schemat szyfrowania hybrydowego, inspirowany HPKE[33], w którym statyczny klucz X25519 wykorzystywany jest do wyznaczenia wspólnego sekretu, a następnie ChaCha20-Poly1305 służy do uwierzytelnionego szyfrowania wiadomości przy użyciu losowego nonce dla każdej wiadomości. Moduł `crypto/storage.rs` odpowiada za szyfrowanie wrażliwych danych przechowywanych w bazie, wykorzystując algorytm Argon2id[34] do derywacji klucza z hasła użytkownika.

Moduł sieciowy (network/) implementuje warstwę komunikacji P2P opartą na bibliotece libp2p. Centralnym elementem jest plik `network/behaviour.rs`, w którym definiowany jest złożony behaviour libp2p łączący protokoły czatu i mailboxa (request–response), mechanizmy odkrywania peerów (mDNS oraz Kademlia DHT) oraz pingo-

[32]Ed25519 to schemat podpisu cyfrowego oparty na krzywej eliptycznej Curve25519, zaprojektowany dla wysokiego bezpieczeństwa oraz wydajności. Wykorzystywany do uwierzytelniania tożsamości węzłów w protokołach P2P, w tym w libp2p, oraz stanowiący praktyczną realizację współczesnych koncepcji kryptografii podpisów (por. Katz, Lindell, *Introduction to Modern Cryptography*) [33].

[33]HPKE (Hybrid Public Key Encryption) to schemat szyfrowania łączący kryptografię asymetryczną (do uzgodnienia klucza) oraz symetryczną (do szyfrowania danych), zapewniający efektywne szyfrowanie wiadomości dla znanego klucza publicznego odbiorcy. Specyfikacja HPKE została opisana w RFC 9180 (Barnes i in., *Hybrid Public Key Encryption*) [4].

[34]Argon2id to funkcja derywacji kluczy z hasła, będąca wariantem algorytmu Argon2 (zwycięzcy konkursu Password Hashing Competition 2015), łącząca odporność na ataki GPU (Argon2i) oraz ataki pamięci tradeoff (Argon2d). Ze względu na parametryzowalność oraz odporność na ataki sprzętowe jest obecnie rekomendowana jako nowoczesny standard ochrony haseł w systemach praktycznych.

wanie peerów w jeden stos sieciowy. Podmoduł `network/layer/` dostarcza abstrakcję warstwy sieciowej, ukrywając szczegóły implementacyjne libp2p oraz zapewniając prosty interfejs dla wyższych warstw aplikacji. Konfiguracja transportu (TCP zabezpieczone protokołem Noise i multipleksowane za pomocą Yamux) znajduje się w module `net/`. Pliki w katalogu `network/handlers/` implementują obsługę poszczególnych typów zdarzeń sieciowych, w tym `chat.rs` odpowiadający za komunikację między peerami, `mailbox.rs` realizujący protokół przechowywania wiadomości dla użytkowników tymczasowo nieobecnych oraz `discovery.rs` zarządzający procesem wykrywania nowych uczestników sieci.

Moduł persystencji (storage/) enkapsuluje logikę dostępu do bazy danych sled, zapewniając wysokopoziomowe interfejsy dla operacji na danych.
Moduł `storage/friends.rs` zarządza książką adresową kontaktów użytkownika, przechowując identyfikatory peerów, klucze publiczne oraz opcjonalne nazwy przyjazne. Plik `storage/history.rs` implementuje przechowywanie historii konwersacji, wykorzystując klucze kompozytowe składające się z identyfikatorów uczestników konwersacji, znacznika czasowego i nonce, co umożliwia efektywne zapytania zakresowe oraz chronologiczne sortowanie wiadomości. Moduł `storage/outbox.rs` zarządza kolejką wiadomości wychodzących oczekujących na dostarczenie do odbiorców, natomiast logika ponawiania prób z wykorzystaniem wykładniczego cofania z jitterem znajduje się w modułach `sync/backoff.rs` oraz `sync/retry.rs`. Plik `storage/mailbox.rs` obsługuje lokalną skrzynkę wiadomości przechowujących zaszyfrowane wiadomości dla innych użytkowników, których węzły są tymczasowo niedostępne.

Silnik synchronizacji (sync/)
koordynuje procesy asynchroniczne związane z dostarczaniem wiadomości oraz utrzymywaniem spójności stanu. Moduł `sync/engine/discovery/` realizuje okresowe odkrywanie dostępnych węzłów mailbox poprzez zapytania do DHT Kademlia, budując lokalną bazę znanych dostawców usługi. Plik `sync/engine/mailbox/fetch.rs` implementuje mechanizm cyklicznego odpytywania znanych węzłów mailbox o nowe wiadomości adresowane do użytkownika, zapewniając ich pobranie oraz potwierdzenie odbioru

po pomyślnym przetworzeniu. Moduł `sync/engine/outbox/forward.rs` odpowiada za przekazywanie wiadomości z kolejki wychodzących do odpowiednich węzłów mailbox w sytuacji, gdy odbiorca nie jest bezpośrednio dostępny w sieci. Pliki `sync/backoff.rs` oraz `sync/retry.rs` implementują polityki wykładniczego cofania oraz limitów prób retransmisji, co ogranicza liczbę nieudanych prób i pozwala na kontrolowane obciążenie sieci.

Moduł interfejsu przeglądarkowego (web/) realizuje serwer HTTP oraz WebSocket przy użyciu frameworka Axum, udostępniając funkcjonalność systemu poprzez aplikację Vue.js 3.

Plik `web/mod.rs` definiuje trasowanie punktów dostępu REST API umożliwiających wysyłanie wiadomości, pobieranie historii konwersacji oraz zarządzanie listą kontaktów, a także obsługę statycznych plików interfejsu. Moduł `web/api.rs` implementuje procedury obsługi poszczególnych punktów dostępu, wykonując walidację żądań, szyfrowanie i deszyfrowanie danych oraz zwracanie odpowiedzi w formacie JSON. Plik `web/websocket.rs` zarządza połączeniami WebSocket, przekazując zdarzenia systemowe (nowe wiadomości, zmiany statusu peerów) do podłączonych klientów w czasie rzeczywistym, co umożliwia aktualizację interfejsu użytkownika bez konieczności cyklicznego odpytywania serwera[35].

Moduł interfejsu użytkownika terminalowego (ui/) implementuje tryb interaktywny REPL[36] (Read-Eval-Print Loop) oraz TUI[37] (Text User Interface), zapewniając dostęp do głównych funkcji komunikatora w środowisku terminala. Interfejs terminalowy

[35]Cykliczne odpytywanie (ang. *polling*) to technika polegająca na regularnym wysyłaniu przez klienta zapytań do serwera w celu sprawdzenia czy pojawiły się nowe dane. WebSocket eliminuje tę konieczność poprzez umożliwienie serwerowi aktywnego wysyłania danych do klienta w momencie ich dostępności.

[36]REPL (Read-Eval-Print Loop) to interaktywne środowisko programistyczne, w którym interpreter odczytuje polecenia użytkownika, wykonuje je, wyświetla wynik oraz oczekuje na kolejne polecenie. Powszechnie stosowane w językach interpretowanych (np. Python, Node.js) oraz narzędziach administracyjnych i powłokach systemowych.

[37]TUI (Text User Interface) to interfejs użytkownika oparty na tekście, wykorzystujący terminal do wyświetlania znaków oraz symboli ASCII lub Unicode w celu utworzenia interaktywnego interfejsu, będącego alternatywą dla GUI (Graphical User Interface) i niewymagającego środowiska graficznego.

przeznaczony jest głównie dla administratorów węzłów mailbox oraz użytkowników preferujących narzędzia konsolowe.

Zrealizowany system stanowi komunikator peer-to-peer realizujący założone w projekcie funkcjonalności: automatyczne wykrywanie uczestników w sieci lokalnej, uzgadnianie kluczy kryptograficznych i szyfrowanie end-to-end treści wiadomości, obsługę scenariuszy komunikacji asynchronicznej z wykorzystaniem mechanizmu mailbox oraz wykrywanie węzłów mailbox za pomocą DHT Kademlia. System udostępnia interfejs przeglądarkowy oparty na Vue.js 3 oraz interfejs terminalowy. Modularna architektura umożliwia przyszłą rozbudowę o dodatkowe funkcjonalności, takie jak przesyłanie plików, rozmowy grupowe czy integracja z mechanizmami NAT traversal (TURN/STUN), przy zachowaniu podstawowych założeń bezpieczeństwa i decentralizacji.

C2. Efekty realizacji projektu

C2.1. Zrealizowane funkcjonalności

W wyniku realizacji projektu powstała aplikacja typu komunikator peer-to-peer implementująca zestaw funkcjonalności umożliwiających wykrywanie uczestników sieci lokalnej, wymianę zaszyfrowanych wiadomości tekstowych, asynchroniczne dostarczanie wiadomości z wykorzystaniem mechanizmu mailbox oraz korzystanie z interfejsu przeglądarkowego i terminalowego.

Automatyczne wykrywanie uczestników sieci

Mechanizm wykrywania peerów w sieci lokalnej oparty jest na komponencie `libp2p-mdns`, który działa automatycznie i nie wymaga od użytkownika ręcznej konfiguracji adresów IP ani portów sieciowych.

Każdy węzeł przy uruchomieniu rozgłasza swoją obecność w sieci, a jednocześnie nasłuchuje komunikatów mDNS od innych węzłów. Otrzymane informacje są wykorzystywane do aktualizacji listy znanych peerów w warstwie sieciowej oraz do nawiązywania połączeń. Interfejsy użytkownika wykorzystują te informacje do prezentowania aktualnej listy dostępnych połączeń, co pozwala użytkownikowi na komunikację bez konieczności ręcznego wyszukiwania węzłów.

Szyfrowanie end-to-end wszystkich wiadomości

System wykorzystuje wielowarstwowe szyfrowanie zapewniające poufność oraz integralność przesyłanych danych zarówno na poziomie transportu (protokół Noise), jak i aplikacji (schemat oparty na X25519, ChaCha20-Poly1305 i SHA-256, inspirowany standardem HPKE).

Przesyłanie wiadomości tekstowych

System obsługuje wysyłanie oraz odbieranie wiadomości tekstowych kodowanych w UTF-8, co umożliwia wykorzystanie znaków Unicode, w tym znaków alfabetów niełacińskich. Użytkownik może komponować wiadomości zarówno w interfejsie przeglądarkowym, jak i terminalowym. Wiadomości są przesyłane w formacie JSON poprzez protokół request-response zdefiniowany w libp2p; każda wiadomość zawiera znacznik czasowy, identyfikator nadawcy oraz zaszyfrowaną treść. Mechanizm potwierdzeń powoduje, że po pomyślnym przetworzeniu wiadomości przez odbiorcę wysyłane jest potwierdzenie do nadawcy, co umożliwia usunięcie odpowiedniego wpisu z kolejki wiadomości wychodzących oraz aktualizację statusu dostarczenia.

Lokalne przechowywanie historii konwersacji

Wszystkie wysłane oraz odebrane wiadomości zapisywane są w lokalnej bazie danych sled, co umożliwia przeglądanie historii konwersacji po restarcie aplikacji. Baza danych wykorzystuje klucze kompozytowe składające się z identyfikatorów uczestników konwersacji, znacznika czasowego wyrażonego w milisekundach od epoki Unix oraz dodatkowego nonce, co umożliwia efektywne zapytania zakresowe zwracające wiadomości z wybranej konwersacji posortowane chronologicznie. Operacje zapisu są realizowane w sposób atomowy na poziomie pojedynczych rekordów.

Historia konwersacji dostępna jest zarówno w interfejsie przeglądarkowym, jak i terminalowym, z obsługą stronicowania dla wydajnego wyświetlania długich historii. Użytkownik może przeglądać chronologicznie uporządkowane wiadomości, przy czym interfejs rozróżnia wiadomości wysłane oraz odebrane. W historii przechowywane są również podstawowe metadane, takie jak znacznik czasowy oraz status dostarczenia wiado-

mości.

Asynchroniczne dostarczanie wiadomości poprzez mechanizm mailbox

System przechowywania wiadomości dla użytkowników tymczasowo nieobecnych umożliwia komunikację asynchroniczną bez konieczności jednoczesnej obecności obu stron w sieci. W sytuacji gdy odbiorca wiadomości nie jest bezpośrednio dostępny, nadawca lokalizuje dostępne węzły mailbox poprzez zapytania do rozproszonej tablicy haszującej Kademlia. Znalezione węzły mailbox otrzymują zaszyfrowaną wiadomość wraz z identyfikatorem odbiorcy w postaci hasza klucza publicznego; szyfrowanie po stronie nadawcy zapewnia, że operator węzła mailbox nie ma możliwości odczytania treści przechowywanej wiadomości.

Węzły mailbox przechowują wiadomości w swojej lokalnej bazie danych, indeksując je według identyfikatora odbiorcy. Mechanizm przekazywania wiadomości z węzła nadawcy do wielu węzłów mailbox umożliwia replikację wiadomości na co najmniej kilku niezależnych węzłach, co zwiększa dostępność danych pomimo awarii części węzłów przechowujących. Usuwanie starych wiadomości realizowane jest okresowo na podstawie znacznika czasowego, z wykorzystaniem progu wieku zdefiniowanego w konfiguracji.

Po powrocie odbiorcy do sieci jego klient, za pośrednictwem silnika synchronizacji, okresowo odpytuje znane węzły mailbox o oczekujące wiadomości. Po pobraniu wiadomości następuje deszyfrowanie treści przy użyciu klucza prywatnego odbiorcy, zapisanie wiadomości do lokalnej historii konwersacji oraz wyświetlenie jej użytkownikowi. Następnie klient wysyła do węzła mailbox potwierdzenie odbioru, co prowadzi do usunięcia przetworzonych wiadomości z przechowalni.

Wykrywanie dostawców usługi mailbox poprzez DHT

Aby umożliwić zdecentralizowane odkrywanie węzłów oferujących usługę przechowywania wiadomości, zaimplementowano mechanizm ogłaszania oraz wyszukiwania dostawców oparty na rozproszonej tablicy haszującej Kademlia. Węzły deklarujące gotowość do pełnienia roli mailbox publikują ten fakt poprzez wywołanie operacji `start_providing` na DHT z ustaloną wartością klucza identyfikującego usługę mailbox. Implementacja Ka-

demlia w libp2p automatycznie replikuje tę informację na węzłach najbliższych do klucza usługi zgodnie z metryką XOR, zapewniając dostępność informacji pomimo dynamicznych zmian w składzie sieci.

Klienci poszukujący węzłów mailbox dla przesłania wiadomości wykonują zapytanie `get_providers` do DHT, otrzymując w odpowiedzi listę adresów sieciowych węzłów aktualnie oferujących usługę. System przechowuje lokalnie informacje o znanych dostawcach, minimalizując liczbę zapytań do DHT oraz zapewniając szybki dostęp do listy dostępnych węzłów. Okresowy proces odkrywania działający w silniku synchronizacji odświeża listę znanych dostawców, adaptując się do zmian w dostępności węzłów oraz zapewniając aktualność informacji.

Interfejs przeglądarkowy

Oprócz interfejsu terminalowego system udostępnia interfejs przeglądarkowy zbudowany w technologii Vue.js 3. Interfejs komunikuje się z częścią serwerową poprzez REST API dla operacji stanowych oraz WebSocket dla przekazywania zdarzeń w czasie rzeczywistym, co eliminuje konieczność cyklicznego odpytywania serwera o nowe wiadomości. Zmiany stanu systemu, w tym nowe wiadomości, zmiany dostępności kontaktów oraz statusy dostarczenia, są przekazywane do interfejsu poprzez kanał WebSocket.

Interfejs zorganizowany jest w postaci modułowych komponentów Vue wykorzystujących Composition API. Stan aplikacji zarządzany jest centralnie poprzez bibliotekę Pinia. Komponenty interfejsu obejmują listę konwersacji z podglądem ostatnich wiadomości, okno czatu z chronologicznie uporządkowaną historią oraz formularze do dodawania nowych kontaktów oraz komponowania wiadomości.

Interfejs terminalowy

Dla użytkowników preferujących narzędzia konsolowe oraz administratorów węzłów mailbox zaimplementowano interfejs terminalowy oferujący dostęp do głównych funkcji komunikatora w środowisku tekstowym. Interfejs pracuje w trybie REPL, akceptując polecenia użytkownika, wykonując je oraz wyświetlając wyniki w formie tekstowej. Obsługiwane polecenia obejmują między innymi listowanie kontaktów, przeglądanie historii

konwersacji, wysyłanie wiadomości oraz dodawanie nowych znajomych.

Interfejs terminalowy udostępnia także informacje diagnostyczne, w tym logi zdarzeń systemowych oraz informacje o liczbie i liście połączonych peerów, co ułatwia monitorowanie działania systemu.

C2.2. Weryfikacja realizacji celów szczegółowych

Realizacja projektu przebiegała zgodnie z wyznaczonymi celami szczegółowymi określonymi w sekcji założeń projektu. Poniżej przedstawiono szczegółową weryfikację osiągnięcia każdego z celów cząstkowych wraz z opisem zastosowanych rozwiązań technicznych.

Cel: Opracowanie architektury systemu peer-to-peer wykorzystującej bibliotekę libp2p

Architektura systemu została zaprojektowana w oparciu o modułowy stos protokołów libp2p. Centralnym elementem jest komponent `P2PBehaviour` definiowany w pliku `network/behaviour.rs`, łączący w jedno zachowanie protokoły czatu i mailboxa (request-response), mechanizmy odkrywania peerów (mDNS oraz Kademlia DHT) oraz pingowanie peerów. Transport TCP zabezpieczony protokołem Noise i multipleksowany za pomocą Yamux konfigurowany jest w module `net/`. Architektura wyraźnie rozdziela warstwy odpowiedzialności: warstwa sieciowa zajmuje się przesyłaniem danych oraz zarządzaniem połączeniami, warstwa kryptograficzna szyfrowaniem i uwierzytelnianiem, a warstwa aplikacyjna logiką komunikatora.

Cel: Implementacja mechanizmu szyfrowania end-to-end opartego na X25519 oraz ChaCha20-Poly1305

Zaimplementowano wielowarstwowy system szyfrowania łączący kryptografię krzywych eliptycznych z szyfrowaniem symetrycznym uwierzytelnionym. Warstwa transportu zabezpieczona jest protokołem Noise wykorzystującym krzywą Curve25519 do uzgadniania kluczy sesyjnych oraz ChaCha20-Poly1305 do szyfrowania strumienia danych. Dodatkowo każda wiadomość aplikacyjna szyfrowana jest indywidualnie przy użyciu schematu opartego na X25519 i ChaCha20-Poly1305, inspirowanego standardem HPKE,

w którym wspólny sekret wyznaczony przez X25519 wykorzystywany jest do derywacji klucza symetrycznego, a ChaCha20-Poly1305 zapewnia poufność oraz integralność treści przy użyciu losowego nonce dla każdej wiadomości. Implementacja wykorzystuje biblioteki kryptograficzne z ekosystemu RustCrypto oraz dalek-cryptography, a zastosowane algorytmy są zgodne z opisem w dokumentach RFC 7748 oraz RFC 8439.

Cel: Wdrożenie mechanizmu automatycznego wykrywania węzłów w sieci lokalnej z wykorzystaniem mDNS

Protokół Multicast DNS został w pełni zintegrowany z systemem poprzez komponent `libp2p-mdns`, zapewniający automatyczne rozgłaszanie oraz odkrywanie węzłów w obrębie podsieci lokalnej. Każdy węzeł przy starcie publikuje swoje usługi w domenie `.local`, umożliwiając innym węzłom odnalezienie go bez konieczności centralnego rejestru czy ręcznej konfiguracji. System dynamicznie reaguje na zdarzenia mDNS, automatycznie aktualizując listę dostępnych peerów w odpowiedzi na pojawianie się oraz znikanie węzłów w sieci. Użytkownicy mogą opcjonalnie przypisywać pseudonimy kontaktom, co ułatwia identyfikację oraz organizację książki adresowej, przy czym pseudonimy przechowywane są wyłącznie lokalnie oraz nie są rozgłaszane w sieci.

Cel: Stworzenie funkcjonalności umożliwiającej wysyłanie wiadomości tekstowych oraz bezpieczne przechowywanie historii

Zaimplementowano system wymiany wiadomości tekstowych z obsługą kodowania UTF-8. Wiadomości przesyłane są poprzez dedykowany protokół `/chat/1.0.0` wykorzystujący mechanizm request-response libp2p; każda wiadomość zawiera znacznik czasowy, identyfikatory uczestników oraz zaszyfrowaną treść. Mechanizm potwierdzeń powoduje, że po pomyślnym przetworzeniu wiadomości przez odbiorcę nadawca otrzymuje odpowiedź, na podstawie której może zaktualizować status dostarczenia i usunąć wiadomość z lokalnej kolejki wychodzących. Historia konwersacji przechowywana jest w lokalnej bazie danych sled z wykorzystaniem kluczy kompozytowych umożliwiających zapytania chronologiczne według pary uczestników rozmowy. Baza zapewnia atomowość pojedynczych zapisów, a dane są utrzymywane na dysku po synchronizacji. Użytkownik może

opcjonalnie włączyć szyfrowanie danych w spoczynku przy użyciu algorytmu Argon2id, co utrudnia odczytanie historii konwersacji w przypadku fizycznego przejęcia urządzenia.

Cel: Integracja rozproszonego przechowywania wiadomości dla nieobecnych użytkowników z wykorzystaniem DHT

System mailbox został zintegrowany z rozproszoną tablicą haszującą Kademlia, co umożliwia zdecentralizowane przechowywanie wiadomości przeznaczonych dla tymczasowo niedostępnych odbiorców. Węzły oferujące usługę mailbox ogłaszają swoją dostępność poprzez mechanizm `provide` w DHT, a klienci odnajdują je poprzez zapytania `get_providers`, bez konieczności centralnej koordynacji. Wiadomości przechowywane na węzłach mailbox są zaszyfrowane przy użyciu klucza publicznego odbiorcy, co uniemożliwia operatorowi węzła odczytanie ich treści. Mechanizm przekazywania wiadomości do więcej niż jednego węzła mailbox pozwala na replikację wiadomości na wielu niezależnych węzłach, zwiększając dostępność danych pomimo awarii części infrastruktury przechowującej.

Cel: Opracowanie mechanizmu weryfikacji dostarczenia wiadomości oraz autoryzowanego usuwania danych

Zaimplementowano mechanizm potwierdzeń dostarczenia wiadomości obejmujący zarówno przesyłanie bezpośrednie, jak i pośrednie poprzez węzły mailbox. Dla wiadomości dostarczanych bezpośrednio odbiorca, po pomyślnym zdeszyfrowaniu i zapisaniu wiadomości, powoduje wysłanie potwierdzenia do nadawcy, co prowadzi do usunięcia wiadomości z lokalnej kolejki wychodzących oraz aktualizacji statusu dostarczenia w bazie danych i interfejsie użytkownika. W przypadku wiadomości przechowywanych na węzłach mailbox odbiorca, po pobraniu i przetworzeniu wiadomości, wysyła do węzła przechowującego żądanie `ACK` zawierające identyfikator odbiorcy (w postaci hasza klucza publicznego) oraz listę identyfikatorów wiadomości do usunięcia. Węzeł mailbox usuwa wskazane wiadomości ze swojej bazy danych, a ochrona przed nieautoryzowanym usuwaniem opiera się na wykorzystaniu szyfrowanego kanału transportowego oraz nierozgłaszanego wprost identyfikatora odbiorcy.

Cel: Zapewnienie skalowalności systemu poprzez mechanizmy przeciwdziałające przeciążeniom

W celu ograniczenia ryzyka przeciążenia zasobów sieciowych i systemowych zastosowano mechanizmy kontrolujące częstość ponawiania operacji oraz czas oczekiwania na odpowiedzi. Silnik synchronizacji oraz warstwa sieciowa wykorzystują strategię wykładniczego cofania z losowym składowym (exponential backoff with jitter) dla nieudanych prób dostarczenia wiadomości oraz operacji sieciowych, a także ograniczają liczbę kolejnych prób. Dodatkowo konfiguracja protokołów request-response obejmuje limity czasu oczekiwania na odpowiedź, po których operacje są przerywane, co uwalnia zasoby dla innych zadań.

Cel: Zaprojektowanie oraz implementacja interfejsu użytkownika wykorzystującego Vue.js

Zrealizowano interfejs przeglądarkowy w technologii Vue.js 3. Interfejs wykorzystuje Composition API do organizacji kodu komponentów oraz zarządzanie stanem poprzez bibliotekę Pinia. Komunikacja z częścią serwerową realizowana jest dwutorowo: operacje stanowe (wysyłanie wiadomości, pobieranie historii) wykorzystują REST API, natomiast aktualizacje w czasie rzeczywistym (nowe wiadomości, zmiany statusów) przesyłane są poprzez WebSocket. Układ interfejsu został zaprojektowany tak, aby dostosowywać się do dostępnej przestrzeni ekranu.

Cel: Przeprowadzenie testów bezpieczeństwa oraz oceny wydajności systemu

W ramach procesu walidacji przeprowadzono testy ręczne w środowisku wielowęzłowym, obejmującym instancje aplikacji uruchomione na kilku węzłach połączonych wspólną siecią lokalną. Wykorzystywano narzędzie Wireshark do analizy ruchu sieciowego w celu potwierdzenia, że treść wiadomości nie jest przesyłana w postaci niezaszyfrowanej. Testy funkcjonalne obejmowały scenariusze wykrywania peerów, wymiany wiadomości bezpośrednich oraz poprzez mechanizm mailbox, a także sprawdzanie działania cyklicznej synchronizacji. Obserwacje z testów pozwoliły na weryfikację poprawności działania głównych mechanizmów bezpieczeństwa i komunikacji.

C2.3. Analiza działania systemu w praktyce

Działanie systemu komunikacji peer-to-peer można najlepiej zobrazować poprzez szczegółowe prześledzenie scenariuszy użycia od momentu uruchomienia aplikacji przez użytkownika do finalnego dostarczenia wiadomości do odbiorcy. Analiza obejmuje zarówno perspektywę użytkownika obsługującego interfejs, jak i perspektywę techniczną opisującą przepływ danych przez poszczególne warstwy architektury systemu.

Inicjalizacja systemu oraz proces dołączania do sieci

Proces uruchamiania aplikacji rozpoczyna się od załadowania lub wygenerowania tożsamości węzła. System sprawdza obecność pliku `identity.json` w katalogu danych aplikacji; plik ten zawiera zakodowaną parę kluczy Ed25519 wykorzystywaną jako główna tożsamość węzła w sieci libp2p oraz odpowiadający jej klucz X25519 używany do szyfrowania na poziomie aplikacji. W przypadku pierwszego uruchomienia aplikacja generuje nową tożsamość i zapisuje ją w formacie JSON, polegając na mechanizmach ochrony dostępu do plików oferowanych przez system operacyjny. Identyfikator *PeerId* wyprowadzany jest z klucza publicznego Ed25519 zgodnie ze specyfikacją biblioteki `libp2p`.

Po załadowaniu tożsamości następuje inicjalizacja warstwy sieciowej. System konfiguruje nasłuchiwanie na porcie TCP, umożliwiając akceptację przychodzących połączeń od innych węzłów sieci. Warstwa zabezpieczeń wykorzystuje protokół Noise, a następnie aktywowany jest multiplekser yamux[38], pozwalający na równoległe przesyłanie wielu strumieni danych przez jedno połączenie TCP.

Warstwa wykrywania węzłów obejmuje protokół mDNS oraz Kademlia DHT. mDNS odpowiada za rozgłaszanie obecności węzła w sieci lokalnej oraz odbieranie ogłoszeń innych peerów. Kademlia DHT wykorzystywana jest do zarządzania informacjami o dostawcach usług, takich jak mailbox, oraz do obsługi zapytań o dostawców.

Po pomyślnej inicjalizacji warstwy sieciowej uruchamiany jest silnik synchronizacji, odpowiedzialny za cykliczną synchronizację stanu lokalnego z siecią. Silnik w regularnych odstępach czasu wykonuje cykl synchronizacji obejmujący sprawdzenie kolejki

[38] Yamux Specification - pozycja [12] w bibliografii. Multiplekser umożliwiający współdzielenie pojedynczego połączenia sieciowego przez wiele niezależnych strumieni danych.

wychodzących wiadomości, odpytanie znanych węzłów mailbox o nowe wiadomości przychodzące oraz aktualizację listy dostawców usługi mailbox z wykorzystaniem DHT.

Na końcu uruchamiana jest warstwa interfejsu użytkownika. W przypadku interfejsu przeglądarkowego system startuje serwer HTTP Axum nasłuchujący na lokalnym porcie, udostępniający REST API oraz punkt dostępu WebSocket. Statyczne zasoby aplikacji klienckiej Vue.js są osadzone w pliku binarnym aplikacji z użyciem biblioteki `rust_embed`.

Dodawanie kontaktu oraz wymiana kluczy

Użytkownik rozpoczynający komunikację z nowym rozmówcą musi najpierw dodać go do listy kontaktów. W interfejsie przeglądarkowym służy do tego formularz dodawania kontaktu, w którym podawany jest identyfikator PeerId oraz klucz publiczny wykorzystywany do szyfrowania end-to-end. Po stronie serwera żądanie dodania kontaktu jest walidowane, a dane kontaktu (PeerId, klucz E2E oraz opcjonalny pseudonim) zapisywane są w bazie danych w drzewie `friends`. Pseudonimy przechowywane są wyłącznie lokalnie i nie są rozgłaszane w sieci.

Wysyłanie wiadomości do peera dostępnego w sieci

Proces wysyłania wiadomości rozpoczyna się od interakcji użytkownika z interfejsem. W przypadku interfejsu przeglądarkowego aplikacja Vue.js wysyła żądanie HTTP do części serwerowej zawierające identyfikator odbiorcy oraz treść wiadomości. Po stronie serwera żądanie jest walidowane, a następnie treść wiadomości szyfrowana jest przy użyciu schematu opartego na X25519 i ChaCha20-Poly1305. Zaszyfrowana wiadomość jest zapisywana w bazie historii i w kolejce wiadomości wychodzących, a następnie system podejmuje próbę bezpośredniego dostarczenia jej do odbiorcy za pomocą protokołu request-response `/chat/1.0.0`.

Po stronie odbiorcy wiadomość jest odszyfrowywana przy użyciu odpowiadającego klucza prywatnego X25519, a następnie zapisywana w lokalnej historii. Po pomyślnym zapisaniu generowane jest potwierdzenie dostarczenia, które wysyłane jest do nadawcy jako komunikat protokołu `/chat/1.0.0`. Nadawca na podstawie tego po-

twierdzenia aktualizuje status wiadomości oraz może usunąć ją z kolejki wiadomości wychodzących. Dodatkowo część serwerowa propaguje zdarzenie nowej wiadomości oraz aktualizacji statusu do podłączonych klientów przeglądarkowych poprzez kanał WebSocket, co umożliwia aktualizację interfejsu bez przeładowywania strony.

Asynchroniczne dostarczanie poprzez system mailbox

W sytuacji gdy odbiorca jest tymczasowo niedostępny, wiadomość pozostaje w kolejce wychodzącej, a silnik synchronizacji podejmuje próbę przekazania jej do sieci mailbox. W tym celu na podstawie klucza publicznego odbiorcy obliczany jest jego identyfikator w postaci hasza, wykorzystywany zarówno do oznaczania wiadomości w bazie mailboxa, jak i do wyszukiwania dostawców w DHT. Silnik synchronizacji wybiera dostępnych dostawców mailboxa, szyfruje wiadomość w strukturę `EncryptedMessage` i wysyła żądanie `PUT` protokołu `/mailbox/1.0.0` do jednej lub kilku instancji mailbox, co umożliwia jej replikację w sieci. W przypadku niepowodzeń wykorzystywane są mechanizmy backoff oraz ograniczona liczba ponownych prób.

Odbiór wiadomości z mailbox po powrocie do sieci

Gdy odbiorca uruchamia aplikację po okresie nieobecności, silnik synchronizacji inicjuje proces pobierania oczekujących wiadomości z węzłów mailbox. Na podstawie klucza publicznego użytkownika obliczany jest identyfikator odbiorcy w postaci hasza, który wykorzystywany jest jako klucz w zapytaniach DHT `GET_PROVIDERS` oraz w żądaniach `FETCH` protokołu `/mailbox/1.0.0`. Węzły mailbox wyszukują w swojej bazie wiadomości oznaczone danym identyfikatorem odbiorcy i odsyłają je do klienta w postaci listy zaszyfrowanych wiadomości.

Odbiorca przetwarza każdą pobraną wiadomość poprzez standardowy proces deszyfrowania oraz zapisuje ją w lokalnej historii konwersacji. System śledzi, które wiadomości zostały już przetworzone, wykorzystując dedykowane drzewo `seen` w bazie danych, gdzie kluczem jest identyfikator wiadomości. Mechanizm ten zapobiega powieleniu wiadomości w sytuacji, gdy ta sama wiadomość jest przechowywana na wielu niezależnych węzłach mailbox oraz pobierana więcej niż raz.

Po pomyślnym przetworzeniu wiadomości klient generuje potwierdzenie odbioru zawierające identyfikatory przetworzonych wiadomości i przesyła je do węzłów mailbox poprzez żądania `ACK` protokołu `/mailbox/1.0.0`. Węzły mailbox po otrzymaniu takiego żądania usuwają wskazane wiadomości ze swojej bazy danych. Mechanizmy ponownych prób i backoff stosowane przy operacjach ACK zwiększają szansę, że potwierdzenia dotrą do większości węzłów, natomiast w razie dalszych problemów duplikaty wiadomości są rozpoznawane i odrzucane dzięki mechanizmowi `seen`.

C2.4. Interfejs przeglądarkowy - szczegóły implementacji

Interfejs użytkownika zrealizowany jest jako aplikacja przeglądarkowa wykorzystująca framework Vue.js 3 z Composition API oraz TypeScript. Warstwa serwerowa oparta jest na frameworku Axum dla języka Rust, udostępniającym REST API oraz WebSocket dla komunikacji z warstwą kliencką.

Struktura komponentów

Aplikacja składa się z trzech głównych komponentów. Komponent `ChatWindow` wyświetla okno konwersacji zawierające listę wiadomości z treścią, znacznikiem czasu oraz statusem dostarczenia. Wiadomości pobierane są z serwera z domyślnym limitem 50 elementów, a starsze wiadomości ładowane są na żądanie użytkownika poprzez przycisk lub automatycznie przy przewinięciu do góry listy. Komponent wykorzystuje Intersection Observer API do wykrywania widoczności wiadomości oraz automatycznego wysyłania żądań odczytu.

Komponent `ChatList` wyświetla listę konwersacji jako tabelę o stałej szerokości 320 pikseli. Każdy wiersz zawiera deterministycznie wygenerowany awatar oparty na funkcji skrótu identyfikatora PeerId, pseudonim lub skrócony identyfikator, fragment ostatniej wiadomości oraz znacznik czasowy. Lista posortowana jest według czasu ostatniej wiadomości. Komponent umożliwia wyszukiwanie tekstowe filtrujące wyniki według pseudonimu, identyfikatora PeerId lub treści ostatniej wiadomości.

Komponent `AddFriendModal` wyświetla okno dialogowe z formularzem przyjmującym identyfikator PeerId, klucz publiczny E2E oraz opcjonalny pseudonim. Pola identyfikatora oraz klucza publicznego oznaczone są atrybutem HTML5 `required`. Weryfikacja

poprawności formatów wykonywana jest po stronie serwera.

Zarządzanie stanem

Stan aplikacji zarządzany jest przez bibliotekę Pinia.
Magazyn `conversationsStore` przechowuje mapę konwersacji indeksowaną identyfikatorami PeerId oraz historię wiadomości dla każdej konwersacji. Magazyn `friendsStore` zarządza listą kontaktów użytkownika.

Komunikacja z serwerem

Komunikacja realizowana jest dwutorowo. REST API pod prefixem `/api/v1` obsługuje operacje:
`POST /api/v1/messages` – wysyła wiadomości,
`GET /api/v1/messages/:peer_id` – pobiera historię konwersacji z parametrami `limit` oraz `before/latest`,
`POST /api/v1/friends` – dodaje kontakty.
Format wymiany danych to JSON.

Protokół WebSocket pod adresem `/ws` przesyła zdarzenia w czasie rzeczywistym w formacie JSON.
Zdarzenie `new_message` zawiera dane nowej wiadomości przychodzącej, zdarzenie `message_delivered` informuje o dostarczeniu wiadomości. Połączenie WebSocket przy utracie łączności automatycznie nawiązuje ponowne połączenie po 3 sekundach.

Warstwa prezentacji

Interfejs wykorzystuje bibliotekę 7.css imitującą wygląd systemu Windows 7 oraz ikony z zestawu Oxygen dla środowiska KDE. Ikony wyświetlane są z atrybutem CSS `image-rendering` ustawionym na wartość `crisp-edges`, co zachowuje ostrość pikseli. Zastosowano pastelową paletę kolorów (różowy, lawendowy, fioletowy, błękitny) charakterystyczną dla estetyki frutiger aero. Interfejs posiada stały układ przeznaczony dla komputerów stacjonarnych.

Bezpieczeństwo

Dane użytkownika wyświetlane są poprzez interpolację tekstową Vue, która automatycznie maskuje znaki specjalne HTML. Aplikacja nie wykorzystuje mechanizmu `v-html`. Operacje kryptograficzne wykonywane są wyłącznie po stronie serwera. Serwer nasłuchuje wyłącznie na adresie `127.0.0.1`, uniemożliwiając dostęp z sieci zewnętrznej.

C2.5. Mechanizm szyfrowania w działaniu

Mechanizm szyfrowania stanowi kluczowy element systemu odpowiadający za poufność i integralność komunikacji między węzłami. Implementacja wykorzystuje dwie warstwy ochrony kryptograficznej: protokół Noise na poziomie transportu biblioteki `libp2p` oraz szyfrowanie treści wiadomości na poziomie aplikacji z użyciem X25519 i ChaCha20-Poly1305. Takie podejście ogranicza ryzyko podsłuchu i modyfikacji danych, przy zachowaniu prostoty implementacji i oparciu się na dobrze przetestowanych bibliotekach.

Warstwa transportu - protokół Noise

Pierwszą warstwę ochrony stanowi protokół Noise, stosowany jako mechanizm zabezpieczenia transportu w bibliotece `libp2p`. Funkcja budująca transport tworzy połączenie TCP, które następnie jest opakowane konfiguracją `noise::Config::new(keypair)` oraz multipleksem Yamux. Klucz Ed25519 węzła, z którego wyprowadzany jest identyfikator *PeerId*, służy do powiązania sesji Noise z długoterminową tożsamością węzła.

Wykorzystywana jest domyślna konfiguracja protokołu Noise dostarczana przez `libp2p`, oparta na krzywej Curve25519, szyfrze AEAD ChaCha20-Poly1305[39] oraz funkcji skrótu SHA-256. Przy zestawianiu połączenia protokół generuje efemeryczne klucze sesyjne i na ich podstawie wyprowadza klucze używane do szyfrowania i uwierzytelniania strumienia danych. Szczegóły wymiany kluczy i formatu ramek są w całości zaimplementowane w bibliotece `libp2p-noise`, co upraszcza kod aplikacji i zmniejsza ryzyko błędów implementacyjnych.

[39]ChaCha20-Poly1305 - algorytm szyfrowania uwierzytelnionego (AEAD) łączący szyfr strumieniowy ChaCha20 z kodem uwierzytelniającym Poly1305. RFC 8439 określa jego zastosowanie w protokołach bezpieczeństwa.

Warstwa aplikacji - szyfrowanie wiadomości

Na poziomie aplikacji każda wiadomość szyfrowana jest przy użyciu schematu inspirowanego HPKE, zaimplementowanego w module `crypto/hpke.rs`. Każdy węzeł posiada statyczną parę kluczy X25519 generowaną podczas tworzenia tożsamości i zapisywaną w pliku `identity.json`. Przy wysyłaniu wiadomości nadawca wykorzystuje swój prywatny klucz X25519 oraz publiczny klucz odbiorcy, aby obliczyć wspólny sekret metodą Diffie-Hellmana.

Wspólny sekret jest następnie przetwarzany funkcją skrótu SHA-256, a otrzymany 32-bajtowy wynik służy bezpośrednio jako klucz symetryczny dla algorytmu ChaCha20-Poly1305. Dla każdej operacji szyfrowania generowany jest losowy 96-bitowy nonce z wykorzystaniem kryptograficznie bezpiecznego generatora liczb losowych. Następnie prymityw AEAD ChaCha20-Poly1305 szyfruje bajty wiadomości (tekst zakodowany w UTF-8) i dołącza tag uwierzytelniający. Zaszyfrowane dane są przechowywane w polu `content` struktury `Message`, obok identyfikatora wiadomości, znacznika czasu oraz losowego *nonce*. Implementacja nie wykorzystuje dodatkowych danych uwierzytelnionych (AAD) – integralność dotyczy wyłącznie treści wiadomości.

Proces deszyfrowania oraz weryfikacji

Odbiorca rozpoczyna proces deszyfrowania od wydzielenia wartości *nonce* z zaszyfrowanej wiadomości. Następnie wyznacza wspólny sekret, wykorzystując swój prywatny klucz X25519 oraz publiczny klucz drugiej strony. W zależności od scenariusza publiczny klucz nadawcy jest pobierany z listy znajomych (komunikacja bezpośrednia) lub z pola dołączonego do wiadomości przechowywanej w mailboxie.

Ze wspólnego sekretu obliczany jest 32-bajtowy klucz symetryczny poprzez zastosowanie funkcji SHA-256, a następnie prymityw AEAD ChaCha20-Poly1305 próbuje odszyfrować wiadomość i zweryfikować tag uwierzytelniający. W przypadku niepowodzenia weryfikacji (np. w wyniku modyfikacji szyfrogramu) operacja kończy się błędem, a wiadomość nie jest przekazywana dalej do warstwy aplikacji. Po pomyślnym odszyfrowaniu bajty są interpretowane jako tekst UTF-8 i zapisywane w historii rozmów oraz prezentowane użytkownikowi.

Zarządzanie kluczami oraz persystencja

System stosuje rozdzielenie kluczy na poziomie architektury. Długoterminowa tożsamość węzła opiera się na parze kluczy Ed25519 generowanej przez bibliotekę `libp2p` i wykorzystywanej do wyznaczenia identyfikatora *PeerId* oraz uwierzytelniania w warstwie transportowej Noise. Niezależnie od tego każdy węzeł posiada parę kluczy X25519 wykorzystywaną wyłącznie do uzgadniania kluczy na potrzeby szyfrowania wiadomości aplikacyjnych, co wpisuje się w zasadę rozdzielenia odpowiedzialności[40].

Klucze prywatne przechowywane są w pliku `identity.json` w katalogu danych aplikacji. Plik ten zawiera strukturę JSON z zakodowanym kluczem Ed25519 w formacie binarnym `libp2p` oraz prywatnym kluczem X25519 używanym przez moduł HPKE. Dodatkowo użytkownik ma możliwość włączenia szyfrowania danych przechowywanych w lokalnej bazie (takich jak historia wiadomości czy lista kontaktów) przy użyciu klucza wyprowadzonego z hasła algorytmem derywacji Argon2id[41]. Zmniejsza to skuteczność ataków słownikowych w przypadku uzyskania fizycznego dostępu do plików z danymi, choć nie eliminuje ryzyka w przypadku bardzo słabego hasła.

Na poziomie transportu protokół Noise wykorzystuje efemeryczne klucze generowane na czas zestawiania połączenia, z których wyprowadzane są klucze sesyjne używane do szyfrowania strumienia danych. W warstwie aplikacyjnej do szyfrowania treści wiadomości wykorzystywane są statyczne klucze X25519 powiązane z tożsamością węzła oraz losowe wartości *nonce* dla algorytmu ChaCha20-Poly1305, dzięki czemu każda wiadomość posiada inny szyfrogram nawet przy tym samym kluczu.

Ochrona przed zagrożeniami kryptograficznymi

Architektura kryptograficzna systemu adresuje podstawowe zagrożenia poprzez połączenie szyfrowanego transportu Noise z szyfrowaniem treści wiadomości na pozio-

[40]Rozdzielenie odpowiedzialności (ang. *separation of concerns*) - zasada projektowa polegająca na wydzieleniu różnych aspektów funkcjonalności systemu do odrębnych modułów, minimalizująca zależności oraz ograniczająca zasięg potencjalnych błędów lub naruszeń bezpieczeństwa.

[41]Argon2 - nowoczesny algorytm derywacji kluczy zaprojektowany specjalnie do ochrony haseł, zwycięzca konkursu Password Hashing Competition z 2015 roku. Wariant Argon2id łączy odporność na ataki z użyciem GPU oraz kanałami bocznymi.

mie aplikacji. Poufność postępująca[42] zapewniana jest przede wszystkim przez warstwę transportową: `libp2p-noise` używa efemerycznych kluczy sesyjnych, dzięki czemu kompromitacja długoterminowego klucza Ed25519 nie pozwala wprost na odszyfrowanie wcześniej przechwyconych sesji transportowych. Dodatkowe szyfrowanie wiadomości (X25519 + ChaCha20-Poly1305) zapewnia uwierzytelnioną poufność treści, lecz ze względu na statyczne klucze X25519 nie wprowadza odrębnej poufności postępującej per wiadomość.

Ataki typu powtórzenie ograniczane są na dwóch poziomach. Protokół Noise zabezpiecza strumień transportowy poprzez zastosowanie szyfrów AEAD z licznikowymi parametrami *nonce*, co uniemożliwia ponowne zaakceptowanie tej samej ramki w obrębie danej sesji. Na poziomie aplikacyjnym każda wiadomość posiada unikalny identyfikator (`Uuid`) oraz znacznik czasu, a moduł `seen` przechowuje identyfikatory wiadomości, które zostały już przetworzone. Ponowne otrzymanie wiadomości o tym samym identyfikatorze skutkuje jej zignorowaniem, nawet jeśli dotrze inną ścieżką sieciową.

Ochrona przed atakami typu osoba pośrodku[43] opiera się na uwierzytelnionym handshaku Noise powiązanym z identyfikatorami *PeerId* oraz na tym, że aplikacja używa ustalonego klucza publicznego rozmówcy (*E2E Public Key*) przy szyfrowaniu i deszyfrowaniu wiadomości. Projekt nie implementuje dodatkowych, zautomatyzowanych mechanizmów weryfikacji odcisków palców kluczy, ale umożliwia manualną inspekcję identyfikatorów i kluczy w interfejsach CLI oraz webowym.

C2.6. DHT i wykrywanie dostawców usługi mailbox

Rozproszona tablica haszująca (opisana szczegółowo w podsekcji „Rozproszone tablice haszujące") znajduje zastosowanie w systemie do zdecentralizowanego odkry-

[42]Poufność postępująca (ang. *forward secrecy*, perfect forward secrecy) - właściwość protokołu kryptograficznego zapewniająca, że kompromitacja długoterminowych kluczy nie umożliwia odszyfrowania wcześniej przechwyconych komunikatów zaszyfrowanych kluczami sesyjnymi wyprowadzonymi z kluczy efemerycznych.

[43]Atak typu osoba pośrodku (ang. *man-in-the-middle*, MITM) - rodzaj ataku polegającego na przejęciu kontroli nad komunikacją między dwiema stronami przez atakującego, który może podsłuchiwać, modyfikować lub wstrzykiwać komunikaty, podszywając się pod obie strony.

wania węzłów oferujących usługę mailbox dla użytkowników tymczasowo nieobecnych. System wykorzystuje implementację algorytmu Kademlia dostarczoną przez bibliotekę `libp2p` (`kad::Behaviour` z magazynem `MemoryStore`), używając dedykowanych kluczy do publikowania i wyszukiwania dostawców mailbox.

Proces odkrywania dostawców mailbox

Węzły oferujące usługę mailbox ogłaszają swoją dostępność w DHT, wywołując mechanizm `start_providing` dla dedykowanego klucza identyfikującego usługę mailbox. Klienci uruchamiają zapytania `GET_PROVIDERS` dla tego klucza, a otrzymane wyniki są agregowane po identyfikatorach *PeerId*, co eliminuje duplikaty. Lista znalezionych dostawców jest następnie sortowana według lokalnego rankingu opartego na zebranych metrykach wydajności i niezawodności (liczbie udanych i nieudanych interakcji oraz średnim czasie odpowiedzi), tak aby preferować węzły, które w praktyce sprawdzają się najlepiej.

Utrzymanie oraz odświeżanie rekordów dostawców

Rekordy dostawców publikowane w DHT obsługiwanej przez bibliotekę `libp2p` posiadają ograniczony czas życia określany przez konfigurację algorytmu Kademlia. Mailbox node wywołuje funkcję `start_providing` przy uruchomieniu, a dalsze zarządzanie czasem życia rekordów, ich replikacją oraz propagacją w sieci realizowane jest przez wbudowane mechanizmy DHT. Węzły, które przestają działać, stopniowo znikają z wyników zapytań, gdy ich rekordy wygasną lub zostaną zastąpione nowszymi informacjami.

Obecna implementacja aplikacji nie dodaje własnej warstwy okresowego odświeżania rekordów dostawców ani aktywnej weryfikacji żywotności węzłów przechowujących rekordy. Ewentualne rozszerzenia mogłyby obejmować cykliczne ponowne wywoływanie `start_providing` oraz dodatkowe testy dostępności mailboxów w celu szybszego usuwania niedziałających dostawców z lokalnego widoku sieci.

Proces wstępnego dołączania do sieci DHT

Nowy węzeł dołączający do sieci inicjalizuje zachowanie Kademlia i wywołuje procedurę bootstrapu DHT dostarczaną przez bibliotekę `libp2p`. Implementacja przewiduje możliwość skonfigurowania listy węzłów startowych (bootstrap nodes), które mogą

pełnić rolę punktów wejścia do sieci, jednak domyślna konfiguracja aplikacji nie określa takich węzłów i opiera się głównie na wykrywaniu sąsiadów przez mDNS oraz na stopniowym wypełnianiu tablicy trasowania podczas zwykłych interakcji sieciowych. Szczegóły działania zapytań `FIND_NODE`, sposobu organizacji kubełków oraz polityki replikacji obsługiwane są przez bibliotekę `libp2p` i nie wymagają dodatkowej logiki po stronie aplikacji.

Ranking oraz selekcja dostawców mailbox

System implementuje lokalny mechanizm rankingu dostawców mailbox oparty na obserwowanych metrykach wydajności oraz niezawodności, co umożliwia świadomy wybór węzłów do przechowywania wiadomości. Dla każdego znanego dostawcy przechowywane są statystyki obejmujące liczbę udanych i nieudanych interakcji, liczbę kolejnych niepowodzeń, średni czas odpowiedzi oraz znacznik czasu ostatniej pomyślnej operacji. Na tej podstawie obliczana jest punktacja zawierająca się w przedziale od 0 do 1, uwzględniająca m.in. współczynnik sukcesów, świeżość ostatniego sukcesu, szybkość odpowiedzi oraz kary za kolejne niepowodzenia i aktywny backoff.

Przy przekazywaniu wiadomości do mailboxów silnik synchronizacji sortuje dostępnych dostawców według wyliczonej punktacji i próbuje zapisać wiadomość kolejno u najlepiej ocenianych węzłów. Każda operacja `PUT` aktualizuje metryki wydajności i może prowadzić do czasowego „zapomnienia” węzła, jeśli liczba błędów przekroczy ustalone progi. Domyślnie system dąży do zapisania wiadomości co najmniej u dwóch niezależnych dostawców, przy jednoczesnym ograniczeniu liczby prób w jednym cyklu wysyłki. Gdy liczba dostępnych mailboxów jest niewielka, okresowe zapytania DHT o dostawców uzupełniają lokalną listę kandydatów.

C2.7. Porównanie zrealizowanego systemu z początkowymi założeniami

Analiza porównawcza finalnego systemu z założeniami sformułowanymi na etapie projektowania pozwala ocenić stopień realizacji celów oraz zidentyfikować obszary, w których rzeczywista implementacja została rozszerzona względem pierwotnych planów lub pozostaje otwarta na dalszy rozwój. Zasadniczym celem było zapewnienie zdecentralizowanej komunikacji typu peer-to-peer z szyfrowaniem end-to-end oraz obsługą wiado-

mości dla tymczasowo nieobecnych użytkowników przy użyciu węzłów typu mailbox.

Architektura peer-to-peer oparta na bibliotece `libp2p` została zaimplementowana zgodnie z założeniami, z wykorzystaniem modularnego stosu protokołów obejmującego warstwę transportową Noise oraz dedykowane zachowania dla komunikacji czatowej, obsługi mailboxów i odkrywania węzłów. Początkowe założenie wykorzystania protokołu Noise do zabezpieczenia warstwy transportowej zostało zrealizowane, a w trakcie implementacji podjęto dodatkowo decyzję o wprowadzeniu drugiej warstwy szyfrowania na poziomie aplikacji poprzez schemat inspirowany HPKE, co wzmacnia obronę wielowarstwową.

Mechanizm automatycznego wykrywania węzłów w sieci lokalnej poprzez mDNS został zrealizowany zgodnie z planem i pozwala na komunikację bez ręcznej konfiguracji adresów. Jednocześnie jego skuteczność zależy od konfiguracji sieci i może być ograniczona w środowiskach, w których ruch multicast jest filtrowany, co jest zgodne z założeniem, że głównym scenariuszem użycia są sieci domowe i małe sieci biurowe.

System przechowywania wiadomości dla nieobecnych użytkowników został zrealizowany z wykorzystaniem rozproszonej tablicy haszującej Kademlia dostarczanej przez bibliotekę `libp2p`. Węzły mailbox ogłaszają swoją dostępność w DHT, a klienci odkrywają dostawców usługi przy użyciu zapytań `GET_PROVIDERS`. Wewnątrz aplikacji zaimplementowano dodatkowe mechanizmy, takie jak replikacja wiadomości u co najmniej dwóch niezależnych dostawców, lokalny ranking mailboxów oparty na statystykach powodzeń, porażek i czasu odpowiedzi oraz wykładniczy mechanizm cofania z losowym odchyleniem dla obsługi błędów. Mechanizmy te zwiększają praktyczną niezawodność i odporność systemu na niestabilność poszczególnych węzłów.

W obszarze interfejsu użytkownika finalna implementacja wykracza poza pierwotnie zakładany interfejs terminalowy w postaci pętli poleceń tekstowych (REPL). Dodano przeglądarkowy interfejs graficzny zbudowany w technologii Vue 3, komunikujący się z backendem poprzez API HTTP. Oba interfejsy korzystają z tych samych prymitywów kryptograficznych oraz tej samej warstwy sieciowej, dzięki czemu rozszerzenie warstwy prezentacji nie wymagało zmian w modelu zaufania ani mechanizmach szyfrowania.

Mechanizmy związane z niezawodnością i obciążeniem obejmują ograniczenie liczby przechowywanych wiadomości na użytkownika, okres retencji wiadomości w mailboxach, śledzenie statystyk wydajności dla dostawców oraz selektywne „zapominanie" węzłów, które wielokrotnie zawodzą. Rozwiązania te mają charakter praktycznych usprawnień zwiększających odporność systemu na błędy sieciowe i niestabilne węzły, choć nie stanowią formalnie udowodnionego schematu skalowania dla bardzo dużych sieci.

Testowanie systemu miało przede wszystkim charakter funkcjonalny i integracyjny, obejmując scenariusze komunikacji z wykorzystaniem wielu węzłów, węzłów mailbox oraz obsługi opóźnionych dostarczeń. W repozytorium nie utrwalono odrębnego zestawu testów wydajnościowych ani formalnych analiz bezpieczeństwa, co pozostawia przestrzeń na dalsze, bardziej systematyczne badania tych aspektów w przyszłych pracach.

Podsumowując, zrealizowany system realizuje kluczowe założenia projektowe związane z architekturą peer-to-peer, szyfrowaniem end-to-end oraz obsługą mailboxów, jednocześnie wprowadzając dodatkowe mechanizmy poprawiające niezawodność i użyteczność, takie jak druga warstwa szyfrowania oparta na schemacie HPKE, ranking i backoff dostawców mailbox oraz interfejs przeglądarkowy. Szczegółowe mechanizmy przechodzenia przez translację adresów sieciowych, wsparcie komunikacji grupowej czy sformalizowane testy wydajnościowe i bezpieczeństwa pozostają w obecnej wersji poza zakresem implementacji i są omówione w sekcji poświęconej ograniczeniom oraz możliwym kierunkom rozwoju.

C2.8. Ograniczenia obecnej implementacji oraz możliwe kierunki rozwoju

Pomimo że zrealizowany system stanowi funkcjonalną implementację zdecentralizowanego komunikatora peer-to-peer, istnieje szereg ograniczeń wynikających z decyzji projektowych oraz priorytetyzacji zakresu pracy. Identyfikacja tych ograniczeń wraz z możliwymi kierunkami rozwoju pozwala ocenić potencjał systemu jako fundamentu dla przyszłych rozszerzeń oraz adaptacji do szerszych przypadków użycia.

Istotnym ograniczeniem obecnej implementacji jest brak wsparcia dla przechodzenia przez translację adresów sieciowych (NAT), co utrudnia bezpośrednią komunika-

cję między węzłami znajdującymi się za różnymi routerami. System działa efektywnie w sieciach lokalnych, gdzie mDNS zapewnia wykrywanie węzłów, natomiast komunikacja przez internet wymaga, aby przynajmniej jeden z węzłów posiadał publicznie dostępny adres IP lub odpowiednio skonfigurowane przekierowywanie portów. Możliwym kierunkiem rozwoju jest integracja z protokołami STUN[44] oraz TURN[45] znanymi z ekosystemu WebRTC.

System obsługuje obecnie wyłącznie wymianę wiadomości tekstowych i nie zapewnia mechanizmów przesyłania plików, obrazów ani innych załączników multimedialnych. Rozszerzenie o taką funkcjonalność wymagałoby zaprojektowania transferu porcjami dla większych danych, w tym wznawiania przerwanych transmisji oraz ograniczania zużycia przepustowości i przestrzeni dyskowej na węzłach mailbox, na przykład poprzez limity rozmiaru lub dodatkowe mechanizmy kontroli wykorzystania zasobów.

Komunikacja ograniczona jest do konwersacji jeden-na-jeden, bez wsparcia dla grupowych czatów czy kanałów. Implementacja komunikacji grupowej w architekturze peer-to-peer wymagałaby rozwiązania problemów takich jak zarządzanie członkostwem grupy, dystrybucja kluczy szyfrujących do wszystkich uczestników oraz utrzymanie spójnej historii konwersacji przy zmiennej dostępności węzłów. Możliwym kierunkiem rozwoju jest wykorzystanie kryptografii klucza grupowego i protokołów w rodzaju TreeKEM[46].

Obecna implementacja nie oferuje komunikacji głosowej ani wideo, które są typowymi funkcjami współczesnych komunikatorów. Integracja mediów w czasie rzeczywistym wymagałaby wykorzystania technologii takich jak WebRTC do kodowania, transmi-

[44]STUN (Session Traversal Utilities for NAT) - protokół umożliwiający wykrycie publicznego adresu IP węzła znajdującego się za translacją NAT oraz charakterystyki zastosowanego mechanizmu translacji, opisany w RFC 5389 [8] i szeroko wykorzystywany w systemach komunikacji czasu rzeczywistego.

[45]TURN (Traversal Using Relays around NAT) - protokół zapewniający przekazywanie ruchu przez serwery pośredniczące gdy bezpośrednie połączenie P2P nie jest możliwe, zdefiniowany w RFC 5766 [9] i stosowany jako uzupełnienie STUN w ekosystemie WebRTC.

[46]TreeKEM - protokół zarządzania kluczami grupowymi oparty na strukturze drzewa, zapewniający efektywną rotację kluczy oraz poufność postępującą dla komunikacji grupowej, stanowiący podstawę zarządzania kluczami w standardzie MLS (Messaging Layer Security) rozwijanym w ramach IETF.

sji i dekodowania strumieni audio/wideo oraz mechanizmów adaptacyjnych dostosowujących jakość do dostępnej przepustowości, a także rozwiązania wspomnianych wcześniej problemów związanych z NAT.

System nie implementuje mechanizmów reputacji ani moderacji zapobiegających nadużyciom, takim jak spam czy rozsyłanie złośliwych treści. W architekturze zdecentralizowanej tradycyjne podejścia oparte na centralnej moderacji nie są bezpośrednio dostępne, co skłania do rozważenia alternatywnych rozwiązań, takich jak sieć zaufania[47] czy mechanizmy typu dowód pracy[48] dla nowych wiadomości.

Aplikacja jest obecnie projektowana jako rozwiązanie uruchamiane na komputerach stacjonarnych, bez natywnych aplikacji mobilnych dla systemów iOS czy Android. Rozwój mobilny wymagałby dostosowania architektury do ograniczeń platform mobilnych, w szczególności zarządzania połączeniami sieciowymi w kontekście agresywnego zarządzania energią oraz ograniczeń dla procesów działających w tle. Możliwym podejściem jest hybrydowa architektura, w której aplikacja mobilna deleguje część funkcjonalności do towarzyszącego węzła stacjonarnego działającego jako osobisty węzeł mailbox.

Mechanizmy konsensusu oraz replikacja danych w DHT implementowane są przez bibliotekę libp2p bez dodatkowych wzmocnień odporności na ataki bizantyjskie[49], gdzie złośliwe węzły mogą celowo dostarczać nieprawidłowych informacji. Dla zwiększenia odporności systemu można rozważyć integrację z mechanizmami takimi jak Practical Byzan-

[47]Sieć zaufania (ang. *web of trust*) - zdecentralizowany model ustanawiania zaufania, gdzie użytkownicy oznaczają zaufane kontakty, a system propaguje reputację poprzez relacje społeczne bez potrzeby centralnego autorytetu. Koncepcja ta jest szeroko znana m.in. z ekosystemu PGP/OpenPGP.

[48]Dowód pracy (ang. *proof-of-work*) - mechanizm wymagający wykonania kosztownych obliczeniowo zadań przed wykonaniem operacji, wprowadzający ekonomiczną barierę dla masowych nadużyć takich jak spam. Mechanizm ten został spopularyzowany m.in. w kryptowalutach (np. Bitcoin), ale może być również stosowany w systemach komunikacyjnych jako ograniczenie spamu.

[49]Ataki bizantyjskie (ang. *Byzantine attacks*) - kategoria zagrożeń w systemach rozproszonych, gdzie złośliwe węzły mogą zachowywać się arbitralnie, dostarczając nieprawidłowe informacje lub sabotując protokoły komunikacyjne. Nazwa pochodzi od problemu bizantyjskich generałów, klasycznie omawianego w literaturze poświęconej systemom rozproszonym i konsensusowi, por. Kleppmann, *Designing Data-Intensive Applications*.

tine Fault Tolerance[50] lub wykorzystanie rejestru tożsamości opartego na łańcuchu bloków zapewniającego trudną do sfałszowania rejestrację węzłów.

Podsumowując, zidentyfikowane ograniczenia stanowią naturalne granice zakresu projektu inżynierskiego, jednocześnie zarysowując bogate możliwości przyszłego rozwoju. System w obecnej formie dostarcza solidnego fundamentu dla zdecentralizowanej komunikacji, który może być rozszerzany oraz adaptowany do szerszych zastosowań poprzez implementację opisanych mechanizmów. Każdy z zaproponowanych kierunków rozwoju stanowi niezależny obszar badawczy mogący być przedmiotem kolejnych projektów rozwijających ekosystem.

C3. Użyteczność projektu

Zrealizowany system komunikacji peer-to-peer może być wykorzystywany zarówno jako praktyczne narzędzie, jak i platforma badawczo-edukacyjna. Użyteczność wynika z połączenia decentralizacji, szyfrowania end-to-end oraz braku zależności od pojedynczego, scentralizowanego operatora, przy jednoczesnym zachowaniu relatywnie prostej architektury opartej na bibliotece `libp2p`.

C3.1. Praktyczne scenariusze zastosowań

Zdecentralizowana architektura systemu jest szczególnie przydatna w środowiskach, w których scentralizowane rozwiązania napotykają ograniczenia techniczne lub organizacyjne. Typowym scenariuszem jest komunikacja w sieciach lokalnych pozbawionych routingu do internetu, takich jak segmentowane sieci korporacyjne, izolowane sieci przemysłowe czy infrastruktura tymczasowa (konferencje, wydarzenia). W takich kontekstach mechanizm mDNS umożliwia automatyczne wykrywanie węzłów i rozpoczęcie komunikacji bez konieczności konfigurowania serwerów pośredniczących.

System może być również stosowany jako wewnętrzny komunikator w organizacjach, które chcą utrzymać pełną kontrolę nad infrastrukturą komunikacyjną i danymi. Uruchomienie węzłów wyłącznie w sieci wewnętrznej eliminuje potrzebę korzystania z

[50] PBFT (Practical Byzantine Fault Tolerance) - algorytm konsensusu tolerujący do jednej trzeciej złośliwych węzłów w systemie rozproszonym, zaprojektowany przez Castro i Liskow (1999) i stanowiący punkt odniesienia dla praktycznych protokołów tolerujących awarie bizantyjskie.

usług chmurowych dostawców zewnętrznych, a otwarty kod źródłowy pozwala na niezależną weryfikację bezpieczeństwa i zgodności z politykami organizacji.

Istotnym aspektem jest także wartość edukacyjna. Projekt stanowi przykład praktycznej integracji prymitywów kryptograficznych (krzywe eliptyczne X25519, ChaCha20-Poly1305, schemat inspirowany HPKE, protokół Noise dostarczany przez `libp2p`) oraz mechanizmów systemów rozproszonych (Kademlia DHT, request-response, mailbox). Kod może być wykorzystywany jako baza do ćwiczeń laboratoryjnych, modyfikacji i eksperymentów w ramach kursów z bezpieczeństwa, systemów rozproszonych czy programowania w języku Rust.

C3.2. Grupy docelowe oraz profil użytkowników

System adresuje kilka głównych grup użytkowników o różnych priorytetach. Użytkownicy zaawansowani technicznie oraz entuzjaści prywatności docenią możliwość samodzielnego uruchamiania węzłów, transparentność implementacji i brak zależności od zamkniętych serwisów. Interfejs terminalowy, parametry wiersza poleceń i prosty format danych sprzyjają integracji z narzędziami automatyzującymi i skryptami.

Dla dziennikarzy, organizacji pozarządowych i aktywistów system może być atrakcyjny jako alternatywa dla rozwiązań scentralizowanych, ponieważ eliminuje pojedynczy, centralny serwer, który mógłby stać się celem przejęcia lub wymuszenia ujawnienia danych. Należy jednak podkreślić, że obecna wersja nie zapewnia pełnej anonimizacji metadanych — węzły pośredniczące i operatorzy infrastruktury mogą obserwować identyfikatory *PeerId*, ruch sieciowy i fakt komunikacji między węzłami, nawet jeśli treść wiadomości pozostaje zaszyfrowana.

Środowiska korporacyjne i instytucje przetwarzające dane wrażliwe mogą wykorzystywać system jako komponent wewnętrznej infrastruktury komunikacyjnej, pod warunkiem właściwej konfiguracji i integracji z istniejącymi procesami bezpieczeństwa. Interfejs przeglądarkowy oparty na Vue 3 ułatwia adopcję przez użytkowników nietechnicznych, ale sam w sobie nie gwarantuje zgodności z regulacjami prawnymi – tę ocenę należy przeprowadzać w kontekście konkretnego wdrożenia.

Badacze bezpieczeństwa i kryptografii, a także społeczność edukacyjna, mogą

traktować projekt jako punkt wyjścia do analiz i eksperymentów. Kod źródłowy umożliwia modyfikację parametrów kryptograficznych, wprowadzanie alternatywnych prymitywów (np. postkwantowych) oraz badanie wpływu zmian na własności bezpieczeństwa i wydajności. Modularna struktura ułatwia izolowanie poszczególnych warstw (kryptografia, DHT, warstwa transportowa) do celów dydaktycznych.[51]

C3.3. Porównanie z istniejącymi rozwiązaniami

Analiza miejsca projektu w ekosystemie rozwiązań dla bezpiecznej komunikacji wymaga identyfikacji różnic architektonicznych oraz kompromisów bezpieczeństwa względem dojrzałych alternatyw. Poniższa tabela przedstawia systematyczne porównanie kluczowych cech:

System	**Architektura**	**Szyfrowanie**	**Zaufanie**
Signal	Scentralizowana	Signal Protocol	Operator serwera
Matrix	Federacyjna	Olm/Megolm	Home server
Briar	P2P + Tor	Custom E2EE	Brak (anonimowe)
Tox	P2P (DHT)	NaCl crypto_box	Brak
Projekt	P2P (libp2p)	HPKE + Noise	Brak

Tabela 2: Porównanie architektur systemów komunikacji

Signal Protocol stanowi referencyjną implementację szyfrowania end-to-end, wykorzystywaną przez komunikatory takie jak Signal, WhatsApp oraz Facebook Messenger. Architektura scentralizowana opiera się na serwerach kontrolowanych przez Signal Foundation, które pośredniczą w wymianie wiadomości oraz przechowują koperty dla użytkowników offline. Model ten wprowadza pojedynczy punkt zaufania - użytkownicy są zależni od zapewnień operatora odnośnie braku kompromitacji serwerów oraz nielogowania metadanych komunikacji. Niniejszy projekt eliminuje tę zależność poprzez decentralizację,

[51]Formalna weryfikacja (ang. *formal verification*) - metodyka matematycznego dowodzenia poprawności algorytmów oraz implementacji względem formalnej specyfikacji, szeroko stosowana w kryptografii dla zapewnienia braku błędów logicznych. Przegląd formalnych modeli bezpieczeństwa protokołów kryptograficznych można znaleźć m.in. u Katz, Lindell, *Introduction to Modern Cryptography* [33].

kosztem zwiększonej złożoności wdrożenia oraz braku gwarancji dostępności porównywalnej z profesjonalnie utrzymywaną infrastrukturą.

Matrix implementuje otwarty standard zdecentralizowanej komunikacji w modelu federacyjnym. Niezależni operatorzy utrzymują serwery home komunikujące się wzajemnie w celu przekazywania wiadomości między użytkownikami różnych instancji. Protokoły Olm oraz Megolm zapewniają szyfrowanie end-to-end. Architektura federacyjna wymaga wyboru zaufanej instancji lub utrzymywania własnego serwera (co wymaga kompetencji administracyjnych). Metadata - graf społeczny użytkowników oraz historia członkostwa w pokojach - pozostają widoczne dla operatorów home servers. Projekt zastępuje zależność od serwerów architekturą peer-to-peer z bezpośrednimi połączeniami między użytkownikami, kosztem mniejszego zakresu funkcjonalności oraz braku interoperacyjności z ekosystemem Matrix.

Briar stosuje architekturę peer-to-peer zoptymalizowaną dla środowisk wysokiego ryzyka. Wykorzystuje Tor dla anonymizacji połączeń oraz Bluetooth dla lokalnej wymiany wiadomości w przypadku niedostępności internetu. Podobnie jak projekt, Briar eliminuje serwery centralne oraz stosuje silne szyfrowanie. Skupienie na odporności na cenzurę następuje kosztem użyteczności - interfejs mobilny jest mniej intuicyjny niż w komercyjnych komunikatorach, brak jest wersji desktopowej. Projekt stosuje webowy interfejs oraz implementację w Rust (zamiast Javy), jednocześnie rezygnując z anonymizacji poprzez Tor, co zwiększa podatność na analizę ruchu sieciowego przez przeciwników obserwujących całą sieć.

Tox stanowi najbliższą architektoniczną analogię do projektu - implementuje protokół peer-to-peer dla komunikacji tekstowej, głosowej oraz wideo bez centralnych serwerów. Wykorzystuje DHT opartą na Kademlii do wykrywania węzłów oraz przechodzenia przez translację adresów sieciowych. Ekosystem kliencki Tox charakteryzuje się fragmentacją - wiele niezależnych implementacji zapewnia niespójny poziom stabilności oraz różne interfejsy użytkownika. Protokół kryptograficzny opiera się na starszych prymitywach (`crypto_box` z NaCl), podczas gdy projekt wykorzystuje szyfrowanie oparte na nowocześniejszych prymitywach (Noise w warstwie transportowej `libp2p` oraz schemat

szyfrowania inspirowany HPKE w warstwie aplikacyjnej). Tox wykazuje większą dojrzałość ekosystemu obejmującego wiele platform (w tym urządzenia mobilne), niniejszy projekt koncentruje się na scenariuszach desktopowych.

libp2p oraz IPFS stanowią infrastrukturę wykorzystywaną przez projekt. Biblioteka libp2p dostarcza komponenty komunikacyjne dla sieci P2P, stosowane również w systemach takich jak Ethereum 2.0 czy Filecoin. Projekt demonstruje zastosowanie libp2p poza kontekstem aplikacji blockchain, jako fundament aplikacji komunikacyjnej. Implementacja może służyć jako punkt odniesienia dla społeczności libp2p, umożliwiając identyfikację ograniczeń API oraz potencjalnych kierunków rozwoju biblioteki.

Architektura projektu stanowi kompromis między rozwiązaniami scentralizowanymi zapewniającymi wysoką użyteczność przy wymaganym zaufaniu do operatorów a systemami maksymalizującymi odporność na cenzurę kosztem większej złożoności. Dla użytkowników wymagających większej kontroli nad infrastrukturą i akceptujących ograniczenia funkcjonalne wynikające z decentralizacji system oferuje eliminację zależności od podmiotów trzecich przy zachowaniu podstawowej funkcjonalności komunikatora tekstowego.

C3.4. Wartość badawcza oraz potencjał rozwoju

Implementacja stanowi potencjalną platformę badawczą dla eksploracji problemów komunikacji zdecentralizowanej oraz praktycznego zastosowania kryptografii. Kod źródłowy ilustruje przekład abstrakcyjnych konstrukcji (wymiana kluczy Diffie-Hellmana, szyfrowanie AEAD, mechanizmy protokołu Noise w `libp2p`, schemat szyfrowania inspirowany HPKE) na konkretne operacje biblioteczne. Umożliwia to analizę zależności między modelem bezpieczeństwa a szczegółami implementacyjnymi.

Architektura pozwala również na badania skalowalności systemów P2P. Parametry takie jak rozmiar kubełków Kademlii, częstotliwość zapytań DHT, strategia rankingu mailboxów czy ustawienia mechanizmu backoff mogą być modyfikowane w celu obserwacji wpływu na opóźnienia, obciążenie sieci i odporność na churn[52].

[52]Churn - dynamiczne dołączanie oraz opuszczanie sieci przez węzły, charakterystyczne dla systemów P2P i szeroko analizowane w literaturze dotyczącej systemów rozproszonych i sieci peer-to-peer.

Z uwagi na modularną budowę warstwy kryptograficznej projekt nadaje się do eksperymentów z kryptografią postkwantową, np. przez zastąpienie X25519 i Ed25519 prymitywami opartymi na CRYSTALS-Kyber i CRYSTALS-Dilithium. Może to służyć ocenie wpływu takich zmian na rozmiar kluczy i szyfrogramów oraz koszt obliczeniowy w rzeczywistym systemie.

C4. Autoewaluacja zespołu projektowego

Realizacja projektu zdecentralizowanego systemu komunikacji peer-to-peer stanowiła kompleksowe przedsięwzięcie obejmujące wszystkie etapy cyklu życia oprogramowania, od analizy wymagań poprzez projektowanie architektury oraz implementację po testowanie oraz walidację. Niniejsza sekcja przedstawia refleksję nad procesem realizacji projektu, nabytymi kompetencjami, napotkanymi wyzwaniami oraz osobistym wkładem w poszczególne etapy pracy.

C4.1. Wkład własny w realizację projektu

Projekt został zrealizowany samodzielnie, obejmując zarówno analizę literatury i projekt architektury, jak i implementację techniczną oraz dokumentację. Prace rozpoczęto od przeglądu protokołów kryptograficznych, architektur rozproszonych i istniejących komunikatorów, co pozwoliło zaplanować system w oparciu o współczesne standardy (m.in. X25519, ChaCha20-Poly1305, Kademlia DHT, biblioteka `libp2p`).

Etap projektowania architektury obejmował wybór stosu technologicznego (język Rust, `libp2p`, `sled`, Vue 3) oraz zaprojektowanie modularnej struktury systemu rozdzielającej odpowiedzialności pomiędzy moduły kryptograficzne, warstwę sieciową, warstwę przechowywania danych i interfejs użytkownika. Taki podział ułatwił późniejszą implementację i modyfikacje poszczególnych komponentów.

Implementacja kodu źródłowego była najbardziej czasochłonną częścią pracy i wymagała praktycznego opanowania programowania asynchronicznego w Rust oraz integracji wielu zależności zewnętrznych w spójną całość. Szczególną uwagę poświęcono fragmentom odpowiedzialnym za operacje kryptograficzne (generacja i przechowywanie kluczy, szyfrowanie i deszyfrowanie wiadomości), które były wielokrotnie przeglądane oraz testowane manualnie w scenariuszach komunikacji między węzłami.

Testowanie systemu miało przede wszystkim charakter funkcjonalny i integracyjny. Uruchamiano wiele instancji aplikacji na jednej lub kilku maszynach, sprawdzając wykrywanie peerów, wymianę wiadomości bezpośrednio i przez mailbox oraz zachowanie przy chwilowych błędach sieci. Debugowanie rozproszonych interakcji wymagało wypracowania praktyk związanych z analizą logów, śledzeniem przepływu zdarzeń i systematycznym ograniczaniem zakresu potencjalnych źródeł błędów.

Dokumentacja projektu, obejmująca komentarze w kodzie oraz niniejszą pracę, powstawała równolegle z implementacją. Decyzje projektowe i napotkane problemy były zapisywane na bieżąco, co ułatwiło późniejsze przedstawienie spójnej narracji oraz ograniczyło konieczność odtwarzania szczegółów technicznych po zakończeniu kodowania.

C4.2. Nabyte kompetencje techniczne oraz metodologiczne

Realizacja projektu przyniosła istotny rozwój kompetencji w wielu obszarach inżynierii oprogramowania, bezpieczeństwa komputerowego oraz systemów rozproszonych, gdzie teoretyczna wiedza zdobyta podczas studiów została uzupełniona praktycznym doświadczeniem implementacyjnym.

W zakresie programowania systemowego w języku Rust opanowano zaawansowane koncepcje będące fundamentem bezpieczeństwa oraz wydajności tego języka. Zrozumienie systemu własności oraz mechanizmu wypożyczeń wymagało przełamania nawyków wyniesionych z programowania w językach ze zbieraniem śmieci, gdzie jawne zarządzanie czasem życia obiektów oraz relacjami między nimi stanowiło początkowo istotną barierę poznawczą. Jednak po opanowaniu tych koncepcji stało się oczywiste jak system typów Rust eliminuje całe klasy błędów runtime poprzez weryfikację podczas kompilacji, co radykalnie zwiększa pewność poprawności programu oraz redukuje czas poświęcany na debugowanie problemów z pamięcią.

Programowanie asynchroniczne wykorzystujące środowisko uruchomieniowe Tokio oraz abstrakcje futures stanowiło kolejny obszar intensywnego uczenia się. Zrozumienie jak zadania asynchroniczne są szeregowane przez executor, jak unikać blokowania wątków poprzez właściwe wykorzystanie punktów oczekiwania await, oraz jak komponować złożone przepływy danych z prostych operacji asynchronicznych wymagało studiowania

dokumentacji oraz eksperymentowania z różnymi wzorcami. Nabyta biegłość w programowaniu asynchronicznym okazała się kluczowa dla implementacji warstwy sieciowej, gdzie tysiące równoczesnych połączeń musi być obsługiwanych efektywnie bez tworzenia osobnych wątków systemowych dla każdego połączenia.

Praktyczna kryptografia stanowiła obszar gdzie teoretyczna wiedza z kursów bezpieczeństwa została pogłębiona poprzez rzeczywistą implementację protokołów. Zrozumienie różnicy między abstrakcyjnymi schematami kryptograficznymi opisywanymi w literaturze oraz ich konkretną realizacją przy użyciu bibliotek takich jak dalek-cryptography czy RustCrypto wymagało studiowania dokumentacji standardów IETF oraz analizy przykładowych implementacji. Szczególnie pouczające było poznanie powszechnych pułapek implementacyjnych takich jak ataki czasowe, gdzie nawet teoretycznie bezpieczny algorytm może być kompromitowany przez niestaranną implementację ujawniającą sekrety poprzez obserwowalne różnice w czasie wykonania operacji.

Architektura systemów rozproszonych, szczególnie wykorzystanie rozproszonych tablic haszujących oraz protokołów odkrywania usług, stanowiła nowy obszar wymagający zrozumienia problemów nieobecnych w systemach scentralizowanych. Koncepcje takie jak ewentualna spójność, gdzie różne węzły mogą posiadać różne widoki stanu systemu w danym momencie, czy odporność na awarię części węzłów bez utraty funkcjonalności całości, wymagały przełamania intuicji opartych na programowaniu aplikacji monolitycznych. Implementacja mechanizmu mailbox z wykorzystaniem DHT Kademlia dostarczyła praktycznego doświadczenia jak teoretyczne algorytmy routingu przekładają się na rzeczywiste zapytania sieciowe oraz jak parametry takie jak współczynnik replikacji wpływają na niezawodność oraz overhead systemu.

Integracja frontendu Vue.js z backendem Rust stanowiła okazję do praktyki projektowania interfejsów programistycznych oraz protokołów komunikacji między komponentami systemu. Projektowanie endpointów REST API wymagało przemyślenia jakie operacje powinny być atomowe, jak reprezentować błędy w sposób umożliwiający ich obsługę przez klienta, oraz jak balansować między elastycznością interfejsu oraz prostotą implementacji. Implementacja komunikacji WebSocket dla aktualizacji w czasie rzeczy-

wistym wymagała rozwiązania problemów takich jak reconnection po utracie połączenia, synchronizacja stanu po powrocie online, oraz zapobieganie race conditions gdzie zdarzenia mogą być przetwarzane w innej kolejności niż wystąpiły.

Metodologia pracy nad projektem również uległa znaczącej ewolucji podczas realizacji. Początkowe próby implementacji pełnej funkcjonalności od razu prowadziły do trudnego w debugowaniu kodu zawierającego wiele interakcji między komponentami. Przyjęcie podejścia iteracyjnego, gdzie system był budowany przyrostowo z ciągłym testowaniem każdego dodanego fragmentu funkcjonalności, okazało się znacznie bardziej efektywne. Taka metodologia pozwalała na szybką identyfikację błędów wprowadzonych w ostatnich zmianach oraz zapewniała że w każdym momencie istniała działająca wersja systemu, nawet jeśli o ograniczonej funkcjonalności.

C4.3. Napotkane problemy oraz przyjęte rozwiązania

Realizacja projektu wiązała się z szeregiem wyzwań technicznych i organizacyjnych, których rozwiązanie stanowiło istotną część procesu uczenia się.

Debugowanie systemu rozproszonego, w którym wiele węzłów komunikuje się asynchronicznie, okazało się znacznie trudniejsze niż praca z aplikacją monolityczną. Zastosowano strukturyzowane logowanie oparte na bibliotece `tracing`, w którym kluczowe zdarzenia (połączenia, zapytania DHT, operacje mailbox) są rejestrowane wraz z kontekstem. Analiza logów z wielu instancji równolegle umożliwiła odtwarzanie sekwencji zdarzeń prowadzących do błędów oraz stopniowe eliminowanie typowych problemów, takich jak niespójne stany czy wyścigi zdarzeń.

Integracja z biblioteką `libp2p` wymagała zrozumienia jej modelu zdarzeniowego i kompozycji wielu zachowań (chat, mailbox, discovery, ping) w jednym *NetworkBehaviour*. W praktyce oznaczało to konieczność częstego sięgania do kodu źródłowego biblioteki oraz przykładów projektów korzystających z `libp2p`, a także iteracyjne dostosowywanie konfiguracji i obsługi zdarzeń do potrzeb komunikatora.

Projektowanie warstwy kryptograficznej wymagało rozdzielenia ról poszczególnych kluczy (tożsamość Ed25519 dla transportu `libp2p`, X25519 dla szyfrowania aplikacyjnego) i zadbania o to, aby były używane w odpowiednich miejscach. W efekcie po-

wstały osobne struktury dla zarządzania tożsamością (`Identity`) i kontekstem szyfrowania (`HpkeContext`), co zmniejsza ryzyko przypadkowego użycia niewłaściwego klucza.

Synchronizacja wiadomości poprzez węzły mailbox wymagała zapewnienia powtarzalnych prób dostarczenia przy tymczasowych błędach oraz unikania duplikatów wiadomości. Połączenie mechanizmu ponownych prób z wykładniczym opóźnieniem, śledzenia wydajności mailboxów i rejestrowania przetworzonych wiadomości w dedykowanym drzewie `seen` w bazie `sled` pozwoliło poprawnie obsłużyć przypadki, w których wiadomość trafia do wielu mailboxów lub jest odbierana po dłuższym czasie.

Optymalizacja dostępu do bazy `sled` wymagała zaprojektowania kluczy kompozytowych i struktury drzew tak, aby typowe operacje (historia rozmów, lista znajomych, wyszukiwanie mailboxów) mogły być realizowane za pomocą sekwencyjnych skanów i prefiksów. Proces ten miał charakter iteracyjny: na podstawie obserwacji wydajności i prostoty implementacji dobierano strukturę kluczy i zakresy, z których korzystają odpowiednie moduły storage.

C4.4. Refleksje oraz wnioski z realizacji projektu

Proces realizacji projektu dostarczył szeregu cennych spostrzeżeń dotyczących inżynierii oprogramowania, metodologii pracy oraz osobistego rozwoju kompetencji, które będą kształtować przyszłe podejście do podobnych przedsięwzięć.

Wartość prototypowania oraz iteracyjnego rozwoju okazała się znacznie większa niż pierwotnie zakładano. Początkowa inklinacja do szczegółowego zaprojektowania całego systemu przed rozpoczęciem implementacji prowadziła do sytuacji gdzie teoretyczny projekt napotykał na problemy praktyczne ujawnione dopiero podczas kodowania. Przyjęcie podejścia gdzie minimalny działający prototyp był rozwijany przyrostowo poprzez dodawanie kolejnych warstw funkcjonalności okazało się znacznie bardziej efektywne, gdzie każda iteracja dostarczała działający system umożliwiający walidację założeń oraz wczesne wykrycie problemów architektonicznych wymagających przeprojektowania.

Znaczenie dogłębnego zrozumienia wykorzystywanych bibliotek oraz frameworków stało się oczywiste gdy powierzchowna znajomość interfejsów prowadziła do implementacji nieefektywnych lub błędnych. Inwestycja czasu w studiowanie dokumentacji,

kodu źródłowego oraz przykładów użycia bibliotek takich jak libp2p czy Tokio zwracała się wielokrotnie poprzez uniknięcie pułapek oraz wykorzystanie zaawansowanych możliwości niedostępnych dla użytkowników ograniczających się do podstawowych przykładów. Lekcja ta podkreśla że w nowoczesnej inżynierii oprogramowania, gdzie większość funkcjonalności jest budowana na fundamencie bibliotek zewnętrznych, umiejętność efektywnego uczenia się oraz integracji gotowych komponentów jest równie istotna jak umiejętność pisania kodu od podstaw.

Balans między doskonałością techniczną oraz pragmatyzmem realizacji stanowił istotne wyzwanie projektowe. Naturalna inklinacja do implementacji najbardziej eleganckich oraz teoretycznie optymalnych rozwiązań często kolidowała z ograniczeniami czasowymi oraz zakresem projektu inżynierskiego. Nauka kiedy zaakceptować rozwiązanie wystarczająco dobre zamiast dążyć do ideału stanowi cenną kompetencję dla rzeczywistej pracy inżynierskiej, gdzie projekty muszą być dostarczane w określonych ramach czasowych oraz budżetowych. Jednocześnie obszary krytyczne dla bezpieczeństwa, takie jak implementacja kryptografii, wymagały maksymalnej staranności bez kompromisów, gdzie lekcją jest umiejętność identyfikacji które aspekty systemu wymagają doskonałości, a które mogą być zrealizowane pragmatycznie.

Doświadczenie samodzielnej realizacji kompleksowego projektu obejmującego wszystkie warstwy systemu od niskopoziomowej kryptografii po interfejs użytkownika dostarczyło holistycznego zrozumienia jak różne aspekty inżynierii oprogramowania współdziałają tworząc spójną całość. Wiedza ta jest trudna do zdobycia poprzez kursy akademickie skupiające się na izolowanych zagadnieniach, gdzie projekt stanowił okazję do syntezy kompetencji z różnych dziedzin w kontekście rzeczywistego problemu. Szczególnie wartościowe było doświadczenie podejmowania decyzji architektonicznych gdzie należało rozważyć trade-offs między różnymi aspektami takimi jak bezpieczeństwo, wydajność, łatwość implementacji oraz użyteczność, gdzie nie istnieje jednoznacznie poprawne rozwiązanie lecz seria kompromisów odpowiednich dla konkretnego kontekstu.

Gdyby projekt był realizowany ponownie, kilka obszarów można by zaplanować inaczej. Wcześniejsze przygotowanie środowiska testowego pozwalającego automatycz-

nie uruchamiać wiele węzłów i odtwarzać typowe scenariusze komunikacji mogłoby skrócić czas debugowania i zwiększyć pewność poprawności implementacji. Systematyczne dokumentowanie kluczowych decyzji architektonicznych od początku prac ułatwiłoby późniejsze uzasadnianie przyjętych rozwiązań. Większy nacisk na testy jednostkowe dla krytycznych komponentów, takich jak moduł kryptograficzny i warstwa storage, ułatwiłby refaktoryzację bez ryzyka niezamierzonych regresji.

Plany przyszłego rozwoju obejmują przede wszystkim obszary zidentyfikowane wcześniej jako ograniczenia: mechanizmy NAT traversal umożliwiające komunikację przez internet, obsługę przesyłania plików i załączników multimedialnych oraz eksperymenty z kryptografią postkwantową. Projekt może również stanowić punkt wyjścia dla badań nad skalowalnością systemów P2P i mechanizmami zachęt dla operatorów węzłów mailbox, jeśli system miałby być wykorzystywany w większej skali.

C5. Wykorzystane materiały i bibliografia związana z realizacją projektu

Realizacja projektu opierała się na szerokiej bazie źródeł obejmujących dokumenty standardów branżowych, specyfikacje protokołów, dokumentację techniczną wykorzystywanych bibliotek oraz publikacje naukowe dotyczące kryptografii oraz systemów rozproszonych. Poniżej przedstawiono kluczowe materiały które wpłynęły na decyzje projektowe oraz implementację systemu.

C5.1. Standardy oraz specyfikacje protokołów

1. Langley, A., Hamburg, M., Turner, S. (2016). *RFC 7748: Elliptic Curves for Security*. Internet Engineering Task Force. Dokument definiujący krzywą Curve25519 oraz algorytm wymiany kluczy X25519 wykorzystywane w systemie dla uzgadniania kluczy szyfrujących.

2. Nir, Y., Langley, A. (2018). *RFC 8439: ChaCha20 and Poly1305 for IETF Protocols*. Internet Engineering Task Force. Specyfikacja algorytmu szyfrowania strumieniowego ChaCha20 oraz kodu uwierzytelniającego Poly1305 stanowiących fundament szyfrowania AEAD wykorzystywanego w projekcie.

3. Cheshire, S., Krochmal, M. (2013). *RFC 6762: Multicast DNS*. Internet Engine-

ering Task Force. Opis protokołu mDNS wykorzystywanego dla automatycznego wykrywania węzłów w sieci lokalnej bez konieczności centralnej konfiguracji.

4. Barnes, R., Bhargavan, K., Lipp, B., Wood, C. (2020). *RFC 9180: Hybrid Public Key Encryption*. Internet Engineering Task Force. Opis schematu HPKE stanowiącego punkt odniesienia dla zastosowanego w projekcie uproszczonego schematu hybrydowego szyfrowania na poziomie aplikacji.

5. Fette, I., Melnikov, A. (2011). *RFC 6455: The WebSocket Protocol*. Internet Engineering Task Force. Specyfikacja protokołu WebSocket umożliwiającego dwukierunkową komunikację w czasie rzeczywistym między klientem webowym a serwerem.

6. Fielding, R., Reschke, J. (2014). *RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*.
   Internet Engineering Task Force. Specyfikacja semantyki HTTP definiująca zasady architektury REST wykorzystywanej w interfejsie API projektu.

7. McGrew, D. (2008). *RFC 5116: An Interface and Algorithms for Authenticated Encryption*. Internet Engineering Task Force. Definicja interfejsu AEAD (Authenticated Encryption with Associated Data) wykorzystywanego dla zapewnienia poufności oraz integralności danych.

8. Rosenberg, J., Mahy, R., Huitema, C., Matthews, P. (2008). *RFC 5389: Session Traversal Utilities for NAT (STUN)*. Internet Engineering Task Force. Specyfikacja protokołu STUN służącego do wykrywania zewnętrznych parametrów połączenia węzłów znajdujących się za NAT.

9. Mahy, R., Matthews, P., Rosenberg, J. (2010). *RFC 5766: Traversal Using Relays around NAT (TURN)*. Internet Engineering Task Force. Specyfikacja protokołu TURN wykorzystywanego do przekazywania ruchu przez węzły pośredniczące, gdy bezpośrednie połączenie P2P nie jest możliwe.

C5.2. Specyfikacje frameworków oraz protokołów P2P

10. Perrin, T. (2018). *The Noise Protocol Framework*. Dostępne: `https://noiseprotocol.org/noise.html`. Specyfikacja frameworka Noise definiująca wzorce handshake wykorzystywane dla zabezpieczenia warstwy transportowej komunikacji P2P.

11. Protocol Labs. (2023). *libp2p Specification*. Dostępne: `https://docs.libp2p.io`. Kompleksowa dokumentacja biblioteki libp2p obejmująca architekturę modularnego stosu protokołów dla aplikacji peer-to-peer.

12. Protocol Labs. (2023). *Yamux Specification*. Dostępne: `https://github.com/hashicorp/yamux/blob/master/spec.md`. Specyfikacja multipleksera yamux umożliwiającego współdzielenie pojedynczego połączenia sieciowego przez wiele niezależnych strumieni danych.

C5.3. Publikacje naukowe

14. Maymounkov, P., Mazières, D. (2002). *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*. Proceedings of the 1st International Workshop on Peer-to-Peer Systems. Fundamentalna praca opisująca algorytm Kademlia wykorzystywany w rozproszonej tablicy haszującej projektu.

15. Bernstein, D.J. (2006). *Curve25519: New Diffie-Hellman Speed Records*. Public Key Cryptography - PKC 2006. Wprowadzenie krzywej Curve25519 oraz demonstracja jej właściwości wydajnościowych oraz bezpieczeństwa.

16. Bernstein, D.J. (2008). *ChaCha, a Variant of Salsa20*. Dokument opisujący konstrukcję algorytmu ChaCha20 jako ulepszonej wersji Salsa20 z lepszą dyfuzją oraz odpornością na kryptoanalizę.

C5.4. Dokumentacja techniczna bibliotek oraz narzędzi

18. The Rust Project Developers. (2023). *The Rust Programming Language*. Dostępne: `https://doc.rust-lang.org/book/`. Oficjalna dokumentacja języka Rust obejmująca koncepcje własności, wypożyczeń oraz bezpieczeństwa pamięci.

19. Tokio Contributors. (2023). *Tokio: Asynchronous Runtime for Rust*. Dostępne: `https://tokio.rs`. Dokumentacja środowiska uruchomieniowego Tokio wykorzystywanego dla programowania asynchronicznego w projekcie.

20. Spacejam. (2023). *sled: Modern Embedded Database*. Dostępne: `https://docs.rs/sled`. Dokumentacja bazy danych sled wykorzystywanej dla persystencji danych w systemie.

21. RustCrypto Project. (2023). *RustCrypto: Cryptography in Rust*. Dostępne: `https://github.com/RustCrypto`. Dokumentacja kolekcji bibliotek kryptograficznych implementujących standardowe algorytmy w języku Rust.

22. Dalek Cryptography. (2023). *curve25519-dalek: Pure-Rust Implementation of Curve25519*. Dostępne: `https://github.com/dalek-cryptography/curve25519-dalek`. Dokumentacja biblioteki implementującej operacje na krzywej Curve25519 wykorzystywanej w projekcie.

23. Evan You and Vue Core Team. (2023). *Vue.js 3 Documentation*. Dostępne: `https://vuejs.org`. Oficjalna dokumentacja frameworka Vue.js 3 wykorzystywanego dla implementacji interfejsu webowego.

24. Eduardo San Martin Morote. (2023). *Pinia: The Vue Store*. Dostępne: `https://pinia.vuejs.org`. Dokumentacja biblioteki Pinia wykorzystywanej dla zarządzania stanem aplikacji frontendowej.

25. David Pedersen and Axum Contributors. (2023). *Axum: Web Framework for Rust*. Dostępne: `https://docs.rs/axum`. Dokumentacja frameworka Axum wykorzystywanego dla implementacji serwera HTTP oraz WebSocket.

C5.5. Zasoby internetowe oraz dokumentacja projektów

26. Protocol Labs. (2023). *IPFS Documentation*. Dostępne: `https://docs.ipfs.tech`. Dokumentacja InterPlanetary File System wykorzy-

stującego libp2p, stanowiąca źródło wzorców użycia biblioteki w produkcyjnych systemach.

27. Marlinspike, M., Perrin, T. (2016). *The Double Ratchet Algorithm*. Dostępne: `https://signal.org/docs/`. Dokumentacja protokołu Signal wykorzystywanego jako punkt odniesienia dla właściwości kryptograficznych komunikatorów E2EE.

28. Hodges, A., Birlea, M. (2023). *Briar Project Documentation*. Dostępne: `https://briarproject.org`. Dokumentacja komunikatora Briar stanowiącego porównanie dla architektur P2P zorientowanych na odporność na cenzurę.

29. The Tox Project. (2023). *Tox Protocol Specification*. Dostępne: `https://toktok.ltd/spec.html`. Specyfikacja protokołu Tox wykorzystywanego jako analog projektu dla oceny alternatywnych podejść do komunikacji P2P.

C5.6. Książki oraz monografie

31. Tanenbaum, A.S., Wetherall, D. (2010). *Computer Networks*. Fifth Edition. Prentice Hall. Kompleksowe opracowanie sieci komputerowych obejmujące fundamenty protokołów transportowych oraz aplikacyjnych wykorzystywanych w projekcie.

32. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A. (1996). *Handbook of Applied Cryptography*. CRC Press. Fundamentalne kompendium kryptografii stosowanej stanowiące źródło teoretycznych podstaw dla implementacji mechanizmów bezpieczeństwa.

33. Katz, J., Lindell, Y. (2014). *Introduction to Modern Cryptography*. Second Edition. Chapman and Hall/CRC. Współczesne ujęcie kryptografii obejmujące dowody bezpieczeństwa oraz formalne podejście do projektowania protokołów.

34. Kleppmann, M. (2017). *Designing Data-Intensive Applications* [34]. O'Reilly Media. Opracowanie architektury systemów rozproszonych obejmujące replikację, partycjonowanie oraz spójność danych w kontekście aplikacji rozproszonych.

C5.7. Artykuły oraz zasoby edukacyjne

35. libp2p Community. (2023). *Understanding libp2p*. Series of conceptual articles. Dostępne:
`https://docs.libp2p.io/concepts/`. Seria artykułów edukacyjnych wyjaśniających koncepcje architektury libp2p wykorzystywane w projekcie.

36. Mozilla Developer Network. (2023). *WebSocket API Documentation*. Dostępne:
`https://developer.mozilla.org/en-US/docs/Web/API/WebSocket`. Dokumentacja interfejsu WebSocket wykorzystywanego dla komunikacji w czasie rzeczywistym między frontendem oraz backendem.

37. Rust Community. (2023). *The Cargo Book*. Dostępne:
`https://doc.rust-lang.org/cargo/`. Dokumentacja narzędzia Cargo wykorzystywanego dla zarządzania zależnościami oraz budowania projektu.

Przedstawiona bibliografia obejmuje kluczowe źródła wykorzystane podczas realizacji projektu. Dodatkowo konsultowano dokumentację inline bibliotek dostępną poprzez system dokumentacji Rust (rustdoc) oraz zasoby społeczności open-source dostępne na platformach takich jak GitHub, Stack Overflow oraz specjalistyczne fora dyskusyjne. Wybór źródeł odzwierciedla interdyscyplinarny charakter projektu łączącego kryptografię, systemy rozproszone, inżynierię oprogramowania oraz projektowanie interfejsów użytkownika.

C6. Spis załączników

1. Archiwum `p2p-chat-source.zip` zawierające kompletny kod źródłowy aplikacji (moduł węzła P2P w języku Rust oraz interfejs webowy Vue 3).

2. Dokument `p2p-chat-screenshots.pdf` zawierający zrzuty ekranu interfejsu użytkownika wraz z krótkimi opisami głównych widoków oraz scenariuszy działania aplikacji.

3. Archiwum `p2p-chat-docs.zip` zawierające dokumentację techniczną wygenerowaną narzędziem `cargo doc` na podstawie kodu źródłowego projektu.

4. Dokument `p2p-chat-files-overview.pdf` zawierający opis struktury projektu oraz krótkie omówienie przeznaczenia najważniejszych katalogów i plików źródłowych.

5. Dokument `p2p-chat-cargo-toml.pdf` zawierający szczegółowe omówienie pliku `Cargo.toml`, w tym konfiguracji pakietu, zależności, cech kompilacji oraz skryptów budowania.

6. Dokument `p2p-chat-diagrams.pdf` zawierający proste diagramy architektury systemu, przepływu wiadomości pomiędzy peerami oraz roli modułu mailbox w sieci P2P.