bіМіністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Створення веб-сайту для анонімного листування в реальному часі

Виконав: студент IV курсу, групи СН-41
спеціальності 122 Комп’ютерні науки
(шифр і назва спеціальності)

_____ Васюрина С. Р.
(підпис) (прізвище та ініціали)

Керівник _____ Дмитроца Л. П.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Литвиненко Я. В.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент _____ Карпінський М. П.
(підпис) (прізвище та ініціали)

Тернопіль
2023

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«__» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)
за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)
Студенту Васюрина Сергій Романович
(прізвище, ім'я, по батькові)
1. Тема роботи Створення веб-сайту для анонімного листування в реальному часі

Керівник роботи Дмитроца Леся Павлівна, доцент кафедри комп'ютерних наук
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджені наказом ректора від « 7 » лютого 2023 року № 4/7-133
2. Термін подання студентом завершеної роботи 2023р.
3. Вихідні дані до роботи літературні та інтернет джерела

4. Зміст роботи
Вступ. 1 Аналіз предметної області. 1.1 Проект Tox. 1.2 Клієнти для Linux. 1.3 Мобільні клієнти та клієнт для Windows та iOS. 1.4 Постановка завдання. 2 Проектування веб ресурсу для анонімного листування. 2.1 Концепції створюваного ресурсу. 2.2 Технологія HLS. 2.3 Анонімна маршрутизація. 2.4 Реалізація основних компонентів системи. 2.5 Реалізація мережевої взаємодії кількох клієнтів. 2.6 Реалізація користувацького інтерфейсу. 3 Безпека життєдіяльності, основи охорони праці. 3.1 Актуальність безпеки життєдіяльності. 3.2 Загальні вимоги охорони праці при роботі з ПК. Висновки. Перелік джерел. Додаток А. Реалізація API засобу анонімного листування.
5. Перелік графічного матеріалу
1. Титулка 2. Мета та задачі. 3. Актуальність дослідження. 4. Існуючі засоби для Linux. 5. Існуючі засоби для Windows. 6. Класифікація мереж за ступенем довіри та Технологія HLS. 7. Виконання операції XOR учасниками мережі. 8. Структура HLS. 9. Схема анонімної маршрутизації повідомлень. 10. Алгоритм відправлення та отримання повідомлення. 11. Криптографічні шари та їхні зв'язки. 12. Реалізація основних компонентів системи. 13. Зашифрований пакет. 14. Реалізація мережевої взаємодії кількох клієнтів. 15. Структура роботи сервісу. 16. Користувацький інтерфейс 17. Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці			

7. Дата видачі завдання 23 січня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	23.01.2023	*Виконано*
2.	Підбір джерел по темі роботи та аналіз предметної області.	24.01.2023-26.01.2023	*Виконано*
3.	Проектування веб ресурсу для анонімного листування	27.01.2023-31.01.2023	*Виконано*
4.	Реалізація веб ресурсу для анонімного листування	01.02.2023-07.02.2023	*Виконано*
	Розроблення веб ресурсу для анонімного листування		
5.	Оформлення розділу «Проектування веб ресурсу для анонімного листування»	08.02.2023-09.02.2023	*Виконано*
6.	Оформлення розділу «Реалізація веб ресурсу для анонімного листування»	10.02.2023-12.02.2023	*Виконано*
7.	Виконання завдання до підрозділу «Безпека життєдіяльності»	05.06.2023-06.06.2023	*Виконано*
8.	Виконання завдання до підрозділу «Основи охорони	07.06.2023-08.06.2023	*Виконано*
	праці»		
9.	Оформлення кваліфікаційної роботи	09.06.2023-11.06.2023	*Виконано*
10.	Нормоконтроль	12.06.2023-13.06.2023	*Виконано*
11.	Перевірка на плагіат	14.06.2023	*Виконано*
12.	Попередній захист кваліфікаційної роботи	15.06.2023	*Виконано*
13.	Захист кваліфікаційної роботи	19.06.2023	

Студент _____ Васюрина С. Р.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Дмитроца Л. П.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Створення веб-сайту для анонімного листування в реальному часі // Кваліфікаційна робота освітнього рівня «Бакалавр» // Васюрина Сергій Романович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СН-41 // Тернопіль, 2023 // С. 87, рис. – 46, табл. – , кресл. – , додат. – 1, бібліогр. – 32.

Ключові слова: hls, hlm, golang, bootsrap, jquery.

Кваліфікаційна робота присвячена аналізу та створенню засобу анонімного листування в реальному часі та дослідженню технологій забезпечення анонімної маршрутизації.

В першому розділі кваліфікаційної роботи проведено дослідження предметної області та засобів анонімного листування в реальному часі. Проаналізовано існуючі засоби анонімного спілкування для операційної системи Linux та Windows.

В другому розділі кваліфікаційної роботи досліджено концепцію засобу анонімного листування та проаналізовано технологію HLS. Проведено аналіз можливості анонімної маршрутизації. Також проведено розробку основних компонент для реалізації системи анонімного листування, реалізацію програмних засобів для мережевої взаємодії та спроектовано користувацький інтерфейс засобу анонімного листування.

В третьому розділі проведено детальне дослідження актуальності питання безпеки життєдіяльності у сучасному світі. В процесі дослідження були розглянуті різноманітні аспекти безпеки, пов'язані з використанням ПК та визначено основні загрози. З метою забезпечення безпеки користувачів ПК були сформульовані загальні вимоги щодо охорони праці.

ANNOTATION

Creation of a website for anonymous correspondence in real time // Qualification work of the educational level "Bachelor" // Vasyuryna Serhii Romanovych // Ivan Pulyuy Ternopil National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer of computer sciences, group SN-41 // Ternopil, 2023 // C. 87, fig. – 46, tab. - , armchair. - , add. – 1, bibliography - 32.

Keywords: hls, hlm, golang, bootsrap, jquery.

The qualification work is devoted to the analysis and creation of a means of anonymous correspondence in real time and the research of technologies for providing anonymous routing.

In the first section of the qualification work, a study of the subject area and means of anonymous correspondence in real time was conducted. The existing means of anonymous communication for the Linux and Windows operating systems have been analyzed.

In the second section of the qualification work, the concept of the anonymous correspondence tool was investigated and the HLS technology was analyzed. An analysis of the possibility of anonymous routing was carried out. Also the development of the main components for the implementation of the anonymous correspondence system, the implementation of software tools for network interaction, and the design of the user interface of the anonymous correspondence tool were carried out.

In the third chapter, a detailed study of the relevance of the issue of life safety in the modern world is carried out. During the research, various security aspects related to the use of PCs were considered and the main threats were identified. In order to ensure the safety of PC users, general requirements for occupational health and safety were formulated.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

Tor (англ. The Onion Router) – це браузер, створений для забезпечення анонімності в мережі Інтернет.

ONR (англ. Office of Naval Research) – управління військово-морських досліджень.

EFF (англ. Electronic Frontier Foundation) – міжнародна некомерційна група цифрових прав, розташована в Сан-Франциско, Каліфорнія.

AS (англ. Automation System) – система автоматизації.

TBB (англ. Tor Browser Bundle) – комплект програмного забезпечення, що дозволяє, не входячи в технічні деталі, швидко і легко почати користуватися анонімними мережами проекту Tor.

NRL (англ. Naval Research Laboratory) – дослідницька лабораторія ONL.

HLS (англ. Hidden Lake Service) – ядро прихованої мережі з теоретично доведеною анонімністю.

HLM (англ. Hidden Lake Messenger) – месенджер з використанням HLS.

API (англ. Application programming interface) – спосіб взаємодії двох чи більше комп’ютерних програм одна з одною.

ВСТУП

Актуальність теми. Актуальність розробки анонімного засобу переписки постійно зростає в сучасному цифровому світі. Віртуальна приватність та захист конфіденційної інформації стають все більш важливими з плином часу. В даний час велика кількість особистої інформації передається через електронні канали зв'язку. Люди відправляють особисті повідомлення, фінансові дані, медичну інформацію та інші конфіденційні дані. Анонімний засіб переписки може забезпечити захист цих даних від несанкціонованого доступу і зловживання. Зросла свідомість про важливість особистої приватності в інтернеті. Користувачі хочуть мати контроль над своїми персональними даними і уникнути їх неконтрольованого поширення. Анонімний засіб переписки може дозволити користувачам комунікувати, не розкриваючи свою справжню особистість, і залишати свої дані в таємниці.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи освітнього рівня «Бакалавр» є розробка веб застосунку анонімного листування в реальному часі. Для досягнення поставленої мети потрібно виконати ряд завдань, зокрема:

– Провести дослідження предметної області та засобів анонімного листування в реальному часі.

– Проаналізувати існуючі засоби анонімного спілкування для операційної системи Linux.

– Проаналізувати існуючі засоби анонімного спілкування для операційної системи Windows.

– Розробити концепцію засобу анонімного листування.

– Проаналізувати технологію HLS.

– Дослідити можливості анонімної маршрутизації.

– Провести розробку основних компонент для реалізації системи анонімного листування.

– Провести реалізації програмних засобів для мережевої взаємодії.

– Спроектувати користувацький інтерфейс засобу анонімного листування.

Практичне значення одержаних результатів. Було розроблено архітектуру засобу анонімного листування та реалізовано веб орієнтований додаток з графічним інтерфейсом реалізованих за допомогою bootstrap та jquery.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Проект Tox

У наші неспокійні часи, коли інформація відіграє велику роль, безпечне спілкування в Інтернеті має велике значення. Незважаючи на те, що проект є відносно молодим, він швидко розвивається. Зв'язок між користувачами здійснюється за допомогою додаткового шару над протоколом UDP. Кожному користувачеві надається спеціальний публічний ключ, який використовується для шифрування. Для налагодження з'єднання потрібно підключитися до бекенду (кожен клієнт в мережі може бути бекендом), який може бути вручну вибраний або знайдений автоматично. Також доступна функція пошуку бекендів у локальній мережі. Tox - це не просто месенджер, а цілий протокол обміну інформацією, що базується на роботі пірінгової мережі, подібної до BitTorrent Sync.

Основною перевагою Tox є повна децентралізація та шифрування всього трафіку. Це забезпечує повну анонімність, що є надзвичайно важливим у наш час. Немає єдиного центру ідентифікації користувача. ID користувача створюється і зберігається локально. Код Tox написаний на мові C і поширюється під ліцензією GPLv3. Більшість розробників ніколи не зустрічалися особисто і мешкають на різних континентах. Основними перевагами Tox є відкритий вихідний код, відсутність централізованих серверів і відсутність контролю з боку будь-якої компанії [25].

Для кожної операційної системи розробляються окремі клієнтські додатки. Однак загальна ідея проекту залишається незмінною. Розробники створюють кілька версій клієнтів з різними функціями, проте офіційною вважається найстабільніша і протестована версія. Розробка Tox відбувається за допомогою сервісу GitHub, де можна завантажити найсвіжішу версію вихідних кодів. З'єднання захищене за допомогою проксі-серверів SOCKS, що дозволяє маршрутизувати весь трафік через Tor. Шифрування реалізоване за

допомогою бібліотеки NaCl (читається як "сіль"), розробленої під керівництвом Деніела Бернштайна в університеті штату Іллінойс у Чикаго.

Tox - це не єдиний сервіс захищеного зв'язку. Інші розробники також створюють альтернативи. Наприклад, Briar, створений командою розробників під керівництвом Майкла Роджерса з Делфтського університету, або проект Invisible.im, заснований аналітиком Патріком Греєм та автором фреймворку Metasploit. Ці клієнти є захищеними аналогами WhatsApp, Viber та інших месенджерів. Також існують комерційні рішення для шифрування голосових розмов. Серед найпопулярніших програм можна відзначити Signal для iPhone та Silent Circle для Android [31]. Але Tox може стати альтернативою, яка повністю замінить приватні месенджери та програми для криптографічного зв'язку. На даний момент Tox є безпечним тунелем між вузлами мережі [2].

Щодо розриву відносин з Tox Foundation, то в липні 2015 року розробники Tox оголосили про розрив з Tox Foundation, яка була створена як компанія-представник проекту. Головою та єдиним членом ради директорів Tox Foundation був Шон Куреші (також відомий як Stqism, Alex Straunoff і Nikolai Toryzin). Згідно з повідомленням розробників, Куреші використав частину коштів фонду для особистих цілей. Точна сума, яку він використав, не була вказана, але, за їхніми словами, це було кілька тисяч доларів. Більшість цих коштів була отримана Tox після участі в Google Summer of Code 2014, а інша частина - пожертвування від приватних осіб.

Незважаючи на цей розрив, розробники Tox вирішили продовжити роботу над проектом і змінили домен сайту на tox.chat [5], оскільки Куреші володів попередніми доменами і надавав хостинг. Розробники запевнили, що код проекту не був скомпрометований. Вони також закликали користувачів оперативно змінити репозиторії. Пізніше, у вересні 2015 року, Куреші заявив, що не використовував гроші проекту для особистих цілей, а витратив їх на покриття зростаючих витрат на обслуговування інфраструктури проекту. Він обіцяв надати докази своєї невинності у вигляді чеків та квитанцій про оплату послуг хостингу, але цього так і не зробив.

1.2 Клієнти для Linux

1.2.1 Клієнт uTox

uTox, офіційний клієнт Tox, запропонований розробниками. Зараз, для користувачів Linux є альфа-версія 0.5.0. На жаль, у репозиторіях Ubuntu бінарного пакету uTox немає: проект ще досить нестабільний. Встановлення uTox нескладний процес для досвідченого користувача. Процес встановлення ідентичний в Ubuntu та в Debian.

[ad name="Responbl"]

Все зводиться до додавання до репозиторію Tox, довірчого ключа до нього та встановлення uTox через менеджер пакетів APT [17]. Це працює і для Ubuntu (починаючи з 14.04), і для операційної системи Debian:

/etc/apt/sources.list$CODENAMErelease.

Лістинг команд для для підготовки до становлення приведений нижче на рисунку 1.1.

```
$ echo "deb https://pkg.tox.chat/debian nightly $CODENAME" | sudo tee / etc / apt / sources . list . d / tox . list $
wget - qO - https://pkg.tox.chat/debian/pkg.gpg.key | sudo apt-key add -
$ sudo apt - get install apt - transport - https
$ sudo apt - get update
```

Рисунок 1.1 – Встановлення uTox

У вигляді бінарного пакету uTox доступний користувачам Gentoo та Arch Linux. При необхідності можна зібрати uTox із вихідного коду. Після встановлення uTox необхідно виконати налаштування. Найголовніше в цьому процесі – задати шлях до користувацького профілю Tox, який зберігається локально. Особливу увагу заслуговує TOX ID. Це 76-значне шістнадцяткове число. Випадково згенерований набір байтів, що є унікальним для кожного користувача клієнта. Виглядає TOX ID, так:

41E9CA1A838AB7CA0E825A7C48B90BAFE1E12B9F0467A7AD40A28
21F1344806BD71BCB00A931

Однак є можливість створити зручніший ID. Отримати його можна на сайті uTox. Просто необхідно обрати особі відповідний нік і вставити із додатку свій TOX ID, та отримати зручний та красивий ідентифікатор виду name@utox.org.

При першому запуску uTox запропонує створити новий обліковий запис або ввести дані існуючого акаунту. Графічний Інтерфейс uTox нагадує Skype, лише без реклами. Користуватися програмою досить просто і зручно. Налаштування не мають великої кількості опцій, але досить логічні і зрозумілі будь-якому більш-менш досвідченому користувачеві. Незважаючи на статус альфа-версії, uTox працює досить стабільно. За весь час тестування (більше тижня) він припиняв свою роботу лише декілька раз. Оновлення програмного засобу виходять майже щодня [22].

Головне вікно програмного засобу приведена на рисунку 1.2.

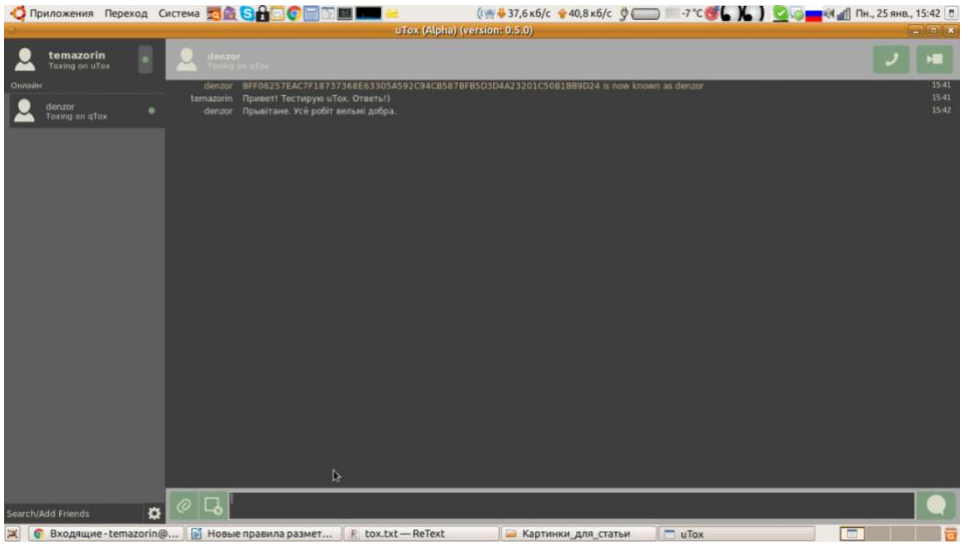


Рисунок 1.2 – Головне вікно uTox

Якість звуку, що передається на рівні SIP. Не варто забувати, що на відміну від Skype у Tox немає інфраструктури серверів. Відеозв'язок теж працює стабільно, без розривів та артефактів. Якість залежить від веб-камери та швидкості з'єднання. Передача невеликих файлів (5-60 Мбайт) відбувається без проблем. С проба відправити відео розміром 160 Мб закінчилася

помилкою. При наступних спробах uTox просто завершує роботу з помилкою. Але це насамперед месенджер, а не засіб пересилання великих файлів. Проект на стадії альфа. І це видно одразу. В цілому, uTox справляє враження практично готового рішення для анонімного спілкування в Мережі. І може практично повністю замінити Skype.

1.2.2 qTox

qTox є наступним "офіційним" клієнтом Tox. Цей програмний продукт написаний на мові програмування C++ з використанням фреймворку Qt 5. Поточна версія на даний момент - 1.2. Установка qTox відбувається аналогічно до установки uTox. За допомогою репозиторію uTox можна встановити і qTox. Розмір повідомлень в qTox обмежений 1372 байтами. Клієнт підтримує аудіо та відеозв'язок, фільтр шуму та придушення ехо, що є корисними функціями, особливо при використанні вбудованого мікрофона та динаміків. Також підтримуються емоджі та проксі, все, як у Skype.

Основною перевагою qTox є його висока швидкість роботи. Цей клієнт Tox для Linux є найшвидшим. Інтерфейс qTox в багатьох аспектах схожий на uTox, але додаток виглядає більш доопрацьованим та зручним для користувача. Крім того, програма рідко вилітає - за весь час вона зламалася всього один раз. Оновлення qTox випускаються щодня, навіть у вихідні дні. Щодо передачі звуку та відео, на мою думку, якість трохи гірша, ніж у uTox, але стабільна. Було б бажано, щоб у майбутніх версіях програми була можливість зміни шрифту в вікні введення тексту. Також було б корисно мати можливість змінювати статус через контекстне меню значка програми в системному треї [23].

Всі клієнти Tox використовують спільну папку профілю для зберігання налаштувань. Однак, дивно, що контакти, додані в список uTox, не відображаються в qTox, і їх доводиться додавати заново до списку контактів.

Це, можливо, пов'язано з тим, що програми мають різні файли налаштувань (рис. 1.3).

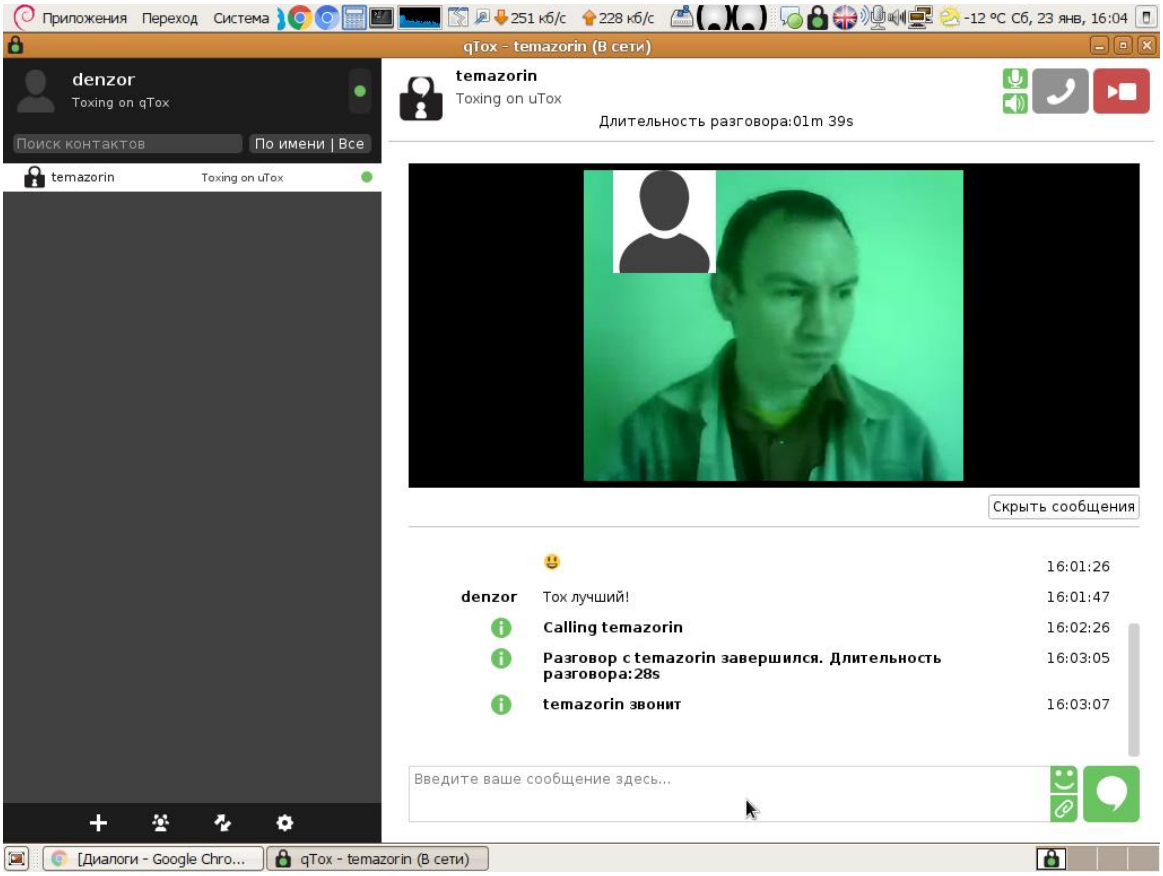


Рисунок 1.3 – Головне вікно qTox

qTox – надає готове рішення для популярного протоколу Tox. Вочевидь, цьому засобу наддадуть перевагу користувачі KDE.

1.2.3 Toxic

Toxic є клієнтом Tox, створеним для справжніх ентузіастів лінуксу. Це консольний клієнт, написаний на мові програмування C з використанням псевдографічної бібліотеки ncurses, і доступний лише для Linux і FreeBSD. Toxic вийшов одним з перших клієнтів Tox, ще в другій половині 2013 року, і є одним з найстаріших. Встановлення Toxic здійснюється через репозиторій Tox, так само, як і у випадку з qTox і uTox. Ті, хто любить виклики, можуть

спробувати зібрати Toxic з вихідних кодів. Для систем BSD існують скомпільовані порти. Використання Toxic досить просте (наскільки це можливо в терміналі) [15]. Для запуску використовується команда "toxic", а для отримання довідки - команда "-". Додаткові параметри програми зберігаються у конфігураційному файлі ".toxic/toxic.conf". Приклад такого файлу можна знайти на сайті Toxic. Головне вікно програмного засобу приведене на рисунку 1.4.

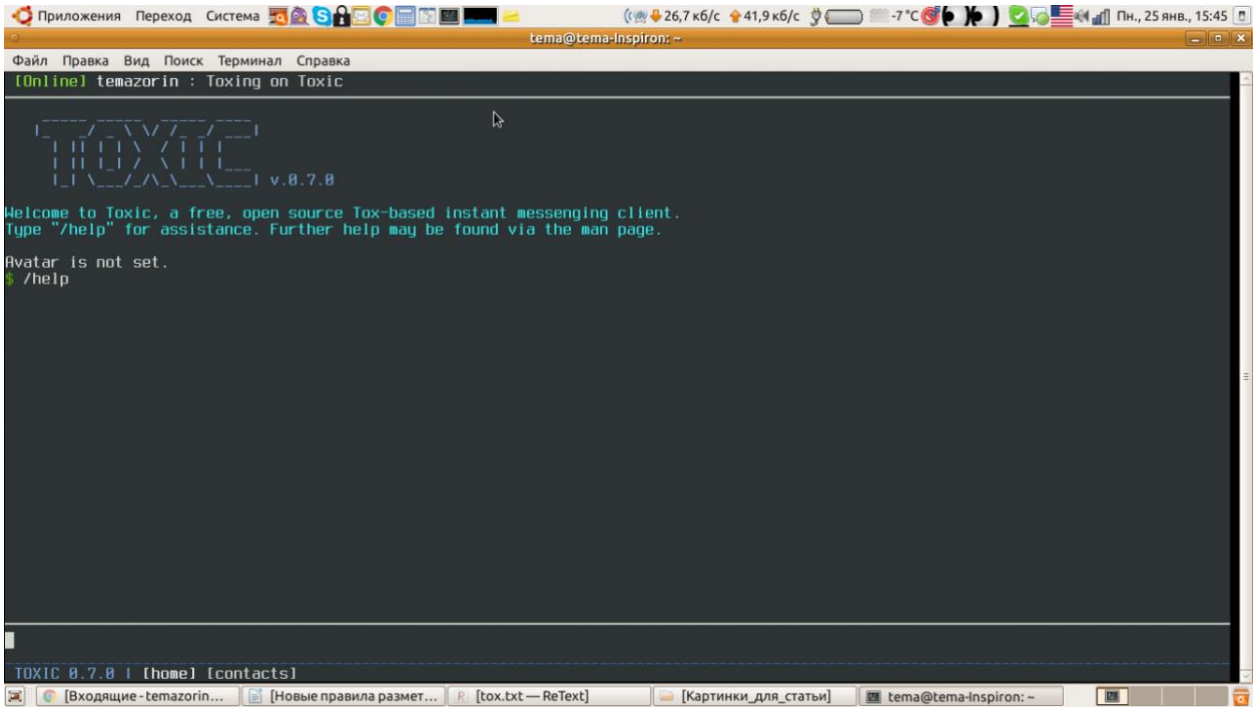


Рисунок 1.4 – Інтерфейс Toxic

Приємно вражає можливість відправляти голосові та навіть відеоповідомлення прямо з робочого столу. З серед основних функцій можна виділити симуляцію статусу "офлайн", підтримку SOCKS5 та HTTP-проксі, блокування небажаних контактів, захист профілів користувачів паролем та шифрування профілів [4]. Звичайно, також підтримується аудіо- та відеозв'язок. За зручністю використання, Toxic надає перевагу над uTox та qTox, але слід пам'ятати, що це консольний клієнт.

1.2.4 Клієнт XwinTox

XwinTox є експериментальним клієнтом Tox, спеціально розробленим для інших BSD-систем, таких як Solaris або FreeBSD, хоча його також можна зібрати в Linux з вихідного коду [19]. Код програми написаний на мовах C і C++, а графічний інтерфейс реалізований за допомогою FLTK - графічного інструментарію. Розробники стверджують, що завдяки модульній структурі, XwinTox є найшвидшим і найбезпечнішим клієнтом Tox. Вони стверджують, що завдяки розділенню на модулі, програма використовує менше ресурсів комп'ютера і працює значно швидше за інших клієнтів Tox (рис. 1.5).

Насправді, в Linux XwinTox працює приблизно так само, як і uTox, хоча він споживає трохи менше пам'яті. Іноді він може відмовляти, особливо при спробі надіслати файл розміром більше 150 МБ.

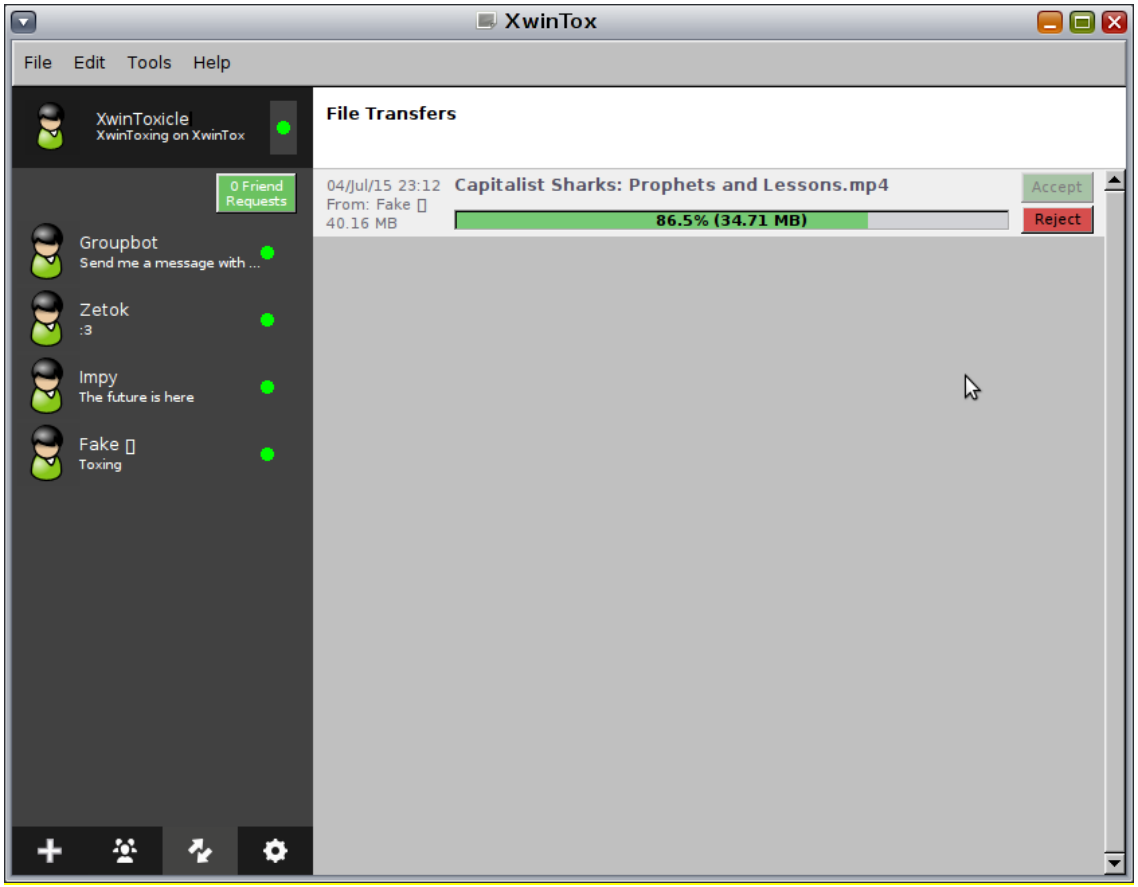


Рисунок 1.5 – Вікно XwinTox

Узагальнюючи, XwinTox можна вважати варіацією uTox, написаною з використанням FLTK замість GTK+ або Qt. Загалом, інтерфейс практично всіх десктопних клієнтів Tox (Linux, OS X або Windows) повторює інтерфейс uTox. Клієнт підтримує обмін текстовими повідомленнями, аудіо- та відеодзвінки, і якість зв'язку не викликає скарг. Проте, це відноситься скоріше до ядра Tox, ніж до самого XwinTox. XwinTox найкраще підходить для користувачів Solaris та BSD-систем.

Однією з найбільших переваг Tox є його повна незалежність від контролю софтверних компаній. Відсутність монополії на Tox є безперечною перевагою, яка дозволяє знищити всі гострі кути та усунути будь-які колишні, поточні та майбутні недоліки Tox. Це велика перемога, оскільки Tox дійсно забезпечує безпеку та анонімність, на які всі чекали.

1.3 Мобільні клієнти та клієнт для Windows та iOS

Програми для операційної системи Windows наразі не є достатньо стабільними та не підтримують усі функції Tox. Їх основне призначення полягає у можливості обміну текстовими повідомленнями та файлами.

Antox є клієнтом Tox для пристроїв з операційною системою Android. Проект активно розробляється та знаходиться на етапі бета-тестування. Наразі користувачам доступний обмін текстовими повідомленнями, файлами та групові чати. Функції аудіо- та відеозв'язку все ще знаходяться у процесі реалізації. Antox можна встановити з репозиторію Google Play Beta, створеного Google спеціально для тестування програм, або зі стороннього репозиторію вільних програм F-Droid. Проте, слід зазначити, що програма наразі знаходиться у відносно сирому стані, і ще рано говорити про повноцінну заміну десктопної версії [24].

Також існує клієнт Tox для iOS під назвою Antidote. Він підтримує обмін текстовими повідомленнями та файлами, а також голосове спілкування. Проте відеозв'язок наразі не є доступним. У додатку присутня функція

шумоподавлення та фільтрації ехо. Розробка проводиться активно, і оновлення виходять дуже часто, іноді навіть кілька разів на добу. Можна очікувати, що найближчим часом всі функції протоколу Tox будуть доступні в цьому додатку.

1.3.1 Клієнт Isotoxin

Зокрема, варто відзначити клієнт Tox для Windows під назвою Isotoxin, який був розроблений з нуля користувачем під псевдонімом Rotkaermota. Ця програма написана на мові програмування C++. Isotoxin вражає своїми можливостями. В ньому повністю реалізована підтримка всіх основних функцій протоколу Tox, включаючи відео дзвінки (рис. 1.6).

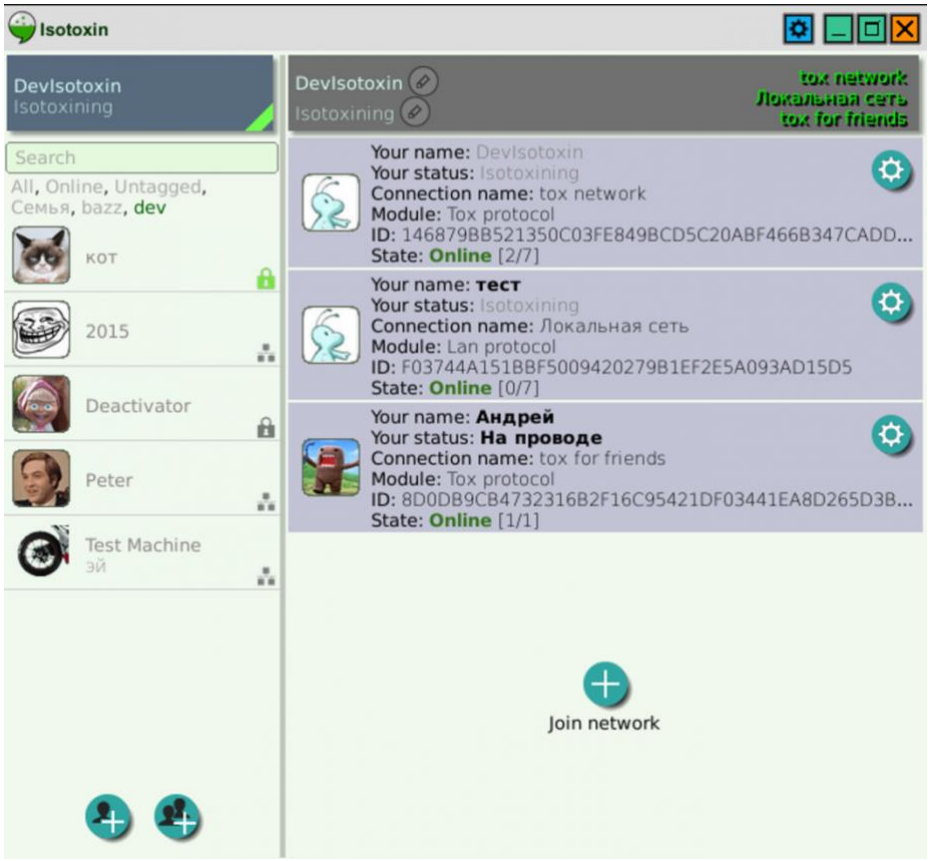


Рисунок 1.6 – Головне вікно додатку Isotoxin

Серед інших функцій варто відзначити власний протокол для спілкування всередині локальної мережі (створений переважно для налагодження системи плагінів, але повністю функціональний: має всі можливості, які є в Tox, за винятком відео), підтримку одночасної роботи з декількома протоколами (наприклад, можна мати два різних підключення до Tox з різними ідентифікаторами), продумані можливості, такі як контакти, аудіо- та відеодзвінки, десктопне захоплення екрану, групові чати, пошук повідомлень, передача файлів, підтримка тем оформлення інтерфейсу тощо [10].

У порівнянні з попереднім дослідженням мало що змінилося. Перевірка на блокування скріншотів була видалена, оскільки це виявилося недостатньо ефективним. Замість цього додано кілька нових пунктів, включаючи можливість використання одного облікового запису на кількох пристроях і перевірку на можливість передачі повідомлень не через інтернет.

Основні критерії порівняння включають:

Вихідний код, доступний за вільною ліцензією (FOSS), і співпраця розробників зі спільнотою, включаючи прийняття патчів.

Ступінь централізації, включаючи наявність центрального сервера, який можна заблокувати, використання мережі серверів або P2P (клієнт-клієнт) архітектури.

Можливість анонімної реєстрації та використання, включаючи прив'язку до телефону та інші методи жорсткої автентифікації.

Наявність захищеного шифрування "від кінця до кінця" (End-to-End Encryption, E2EE). Деякі месенджери мають цю функцію за замовчуванням, інші дозволяють її включити, але деякі зовсім не мають такого шифрування.

Синхронізація зашифрованих чатів "від кінця до кінця". Наявність цієї функції спрощує життя, але її реалізація технічно складна, тому вона не поширена.

Перевірка відбитків E2EE (включаючи групові чати). Не всі месенджери мають функцію перевірки відбитків, а деякі навіть не надають відкритого

доступу до цієї функції. Групові чати без перевірки відбитків перестають бути приватними.

Групові E2EE-чати, які дозволяють шифрувати листування між кількома користувачами.

Можливість використання одного облікового запису на різних пристроях, що дуже зручно.

Захист соціального графа, включаючи збір інформації про контакти користувача та інші дані.

Альтернативні способи передачі даних, які дозволяють використовувати інші шляхи передачі повідомлень, крім інтернету.

1.3.2 Клієнт SafeUM

Під час реєстрації у користувача безкоштовно на 21 день надається анонімний номер телефону з Латвії у форматі +3712XXXXXX.

Щодо шифрування, E2EE (End-to-End Encryption) застосовується за замовчуванням у дзвінках та відеодзвінках, але доступ до E2EE в чатах доступний лише у пакеті Premium. SafeUM також використовує Центр сертифікації ключів, який забезпечує несесійне шифрування і дозволяє верифікувати користувача за його Публічним ключем.

Синхронізація E2EE відбувається шляхом генерації ключів на пристрої користувача під час активації криптографічного чату.

Робота з обліковим записом можлива лише з одного пристрою, тому якщо користувач увійде з іншого пристрою, він автоматично вийде з попереднього.

Для перевірки відбитків, користувач може використовувати QR-код для верифікації Публічного ключа співрозмовника. У групових чатах додаткова перевірка не потрібна, оскільки ключі кожного користувача криптографічного чату вже перевірені заздалегідь.

SafeUM дозволяє вхід з будь-якого пристрою з дозволом користувача операційної системи. При цьому на попередньому пристрої автоматично відбудеться вихід з облікового запису.

Групові E2EE-чати доступні у SafeUM, проте кількість учасників обмежена до 16 осіб через високу енергоємність шифрування, що дозволяє економити заряд батареї користувачів [18].

При зміні ключів шифрування користувачем, всі учасники отримують повідомлення для перевірки справжності співрозмовника.

У груповий чат користувач може бути доданий лише іншим існуючим учасником чату.

За замовчуванням, SafeUM не має доступу до списку контактів. Проте користувач може дозволити програмі доступ до свого списку контактів.

SafeUM не має доступу до геолокації користувача і не пропонує альтернативних методів визначення місцезнаходження (рис. 1.7).

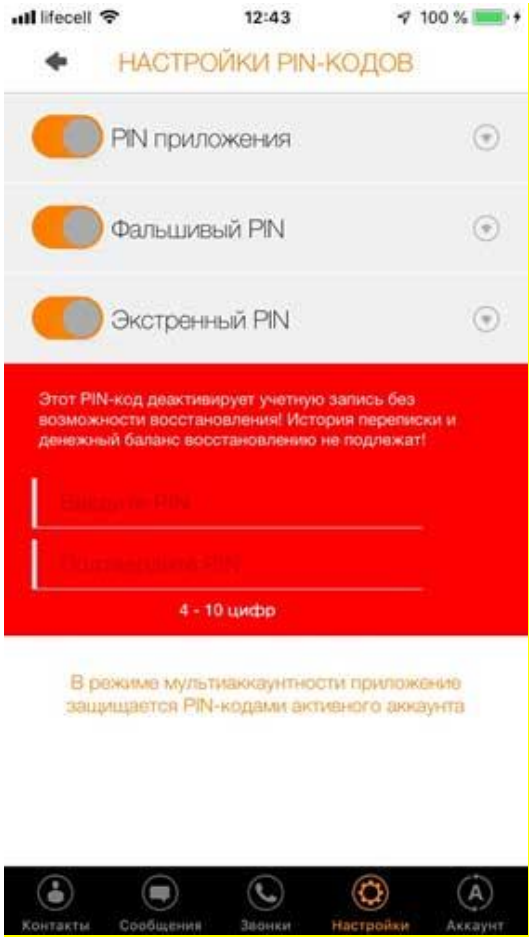


Рисунок 1.7 – Налаштування SafeUM

Використовуються такі алгоритми шифрування: AES-256, SHA-2(256), шифрування та цифровий підпис (ECDSA з SHA256) на еліптичних кривих відповідно до стандарту NIST для підтвердження автентичності сторін зв'язку. Крім того, використовується шифрування з відкритим ключем Ель Гамаля.

Кожен рядок чату динамічно шифрується. Схема шифрування використовує публічні та приватні ключі, які генеруються на основі ключового слова клієнта. Це дозволяє користувачеві працювати з різних пристроїв і мати доступ до шифрованої переписки з будь-якого місця без необхідності експорту-імпорту даних [9].

Історія чату не зберігається на пристрої користувача. Є три PIN-коди з різними рівнями доступу до контенту, що забезпечує додатковий рівень безпеки. Крім того, відсутність реклами та спаму забезпечує комфортне використання. Користувачі також мають можливість здійснювати оплату крипто валютою (рис. 1.8).

Рисунок 1.8 – Набір номеру SafeUM

Розробники програмного засобу сфокусувалися на корпоративному сегменті, що не завадило розвитку проекту та зростанню користувачів.

1.3.3 Клієнт Wickr

Wickr є месенджером, який надає певні функції приватності та безпеки. Основні характеристики Wickr включають:

- Відсутність синхронізації E2EE: Попередні листування не зберігаються при вході в обліковий запис з іншого пристрою, оскільки синхронізація кінцевого до кінця (E2EE) недоступна.

- Перевірка відбитків: Wickr має функцію надсилання короткого відео, що дозволяє співрозмовнику переконатися в автентичності вашої особи. Однак, повідомлення про цю перевірку не надсилається автоматично, і функція може бути скрита в інтерфейсі програми.

- Можливість додавання пристроїв: Ви можете додавати різні пристрої до вашого облікового запису Wickr, що дозволяє вам використовувати месенджер на кількох пристроях одночасно.

- Групові E2EE-чати: Wickr підтримує групові чати з кінцевою до кінця шифруванням, що дозволяє безпечно обмінюватися повідомленнями у груповому форматі.

- Приватність та безпека: Wickr розроблений з урахуванням приватності та безпеки. Листування відбувається через сервери, проте повідомлення автоматично видаляються як з серверів, так і з пристроїв користувача. Користувач може налаштувати тривалість зберігання історії повідомлень [16].

Wickr також привертає увагу до своєї анонімності, проте вказання, що він є "месенджером, заснованим на блокчейні", може бути невірним.

Варто враховувати, що оцінка приватності та безпеки месенджера повинна базуватися на незалежних аудитах безпеки, а також на зрозумілості та надійності протоколів шифрування, що використовуються (рис. 1.9).

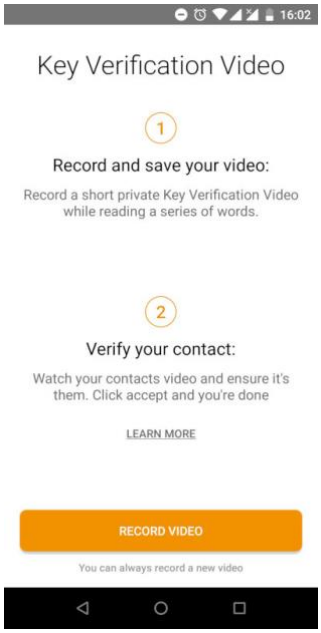


Рисунок 1.9 – Головне вікно Wickr

Wickr забезпечує шифрування всієї передаваної інформації з використанням стандартів, таких як AES-256, ECDH-521 та RSA-4096 TLD. Кожне повідомлення отримує власний ключ для додаткової безпеки. Крім того, всі повідомлення в Wickr не містять імен відправника, одержувача або геолокаційних данних, що забезпечує анонімність користувачів.

На веб-сайті розробника доступні три версії Wickr: "me", "ent" та "pro". Проте безкоштовно завантажити можна лише першу версію, "me", яка призначена для особистого використання. Версії "ent" і "pro" є платними і мають розширений набір функцій.

Цікаво, що Wickr використовується у деяких державних та урядових установах, що свідчить про його довіру та надійність.

1.3.4 Клієнт Tox (Antox)

Клієнт Tox (Antox) є месенджером, який забезпечує захищеність та приватність комунікацій. Деякі особливості цього месенджера включають:

Перевірка відбитків: Для початку діалогу необхідно ввести ідентифікатор співрозмовника або сканувати його QR-код.

Можливість додавання пристроїв: Ви можете імпортувати свій профіль з одного смартфона на інший, що дозволяє використовувати один обліковий запис на кількох пристроях. Також є можливість використовувати групові чати з криптографічним захистом (E2EE).

Повідомлення про перевірку E2EE відсутні: У груповому чаті може долучитись будь-який користувач, який знає його ідентифікатор, без попередньої перевірки відбитків.

Цей месенджер був створений незалежною групою розробників, які приділяють особливу увагу безпеці та приватності. Програма є відкритим джерелом, а всі повідомлення передаються з наскрізним шифруванням, яке неможливо відключити. Інтерфейс месенджера є простим і зрозумілим для користувачів (рис. 1.10).

Рисунок 1.10 – Головне вікно Tox (Antox)

Antox використовує протокол Tox, що дозволяє забезпечити функціонал месенджера, такий як голосовий і відеозв'язок, надсилання миттєвих повідомлень та передача файлів. Також Antox підтримує конференційний

режим з кількома учасниками та інші функції, які характерні для сучасних месенджерів (рис 1.11).

Однак, важливою особливістю Antox є відсутність реклами. Це означає, що вам не будуть відображатись рекламні повідомлення під час користування месенджером. Такий підхід до монетизації дозволяє забезпечити більшу приватність та комфорт для користувачів.

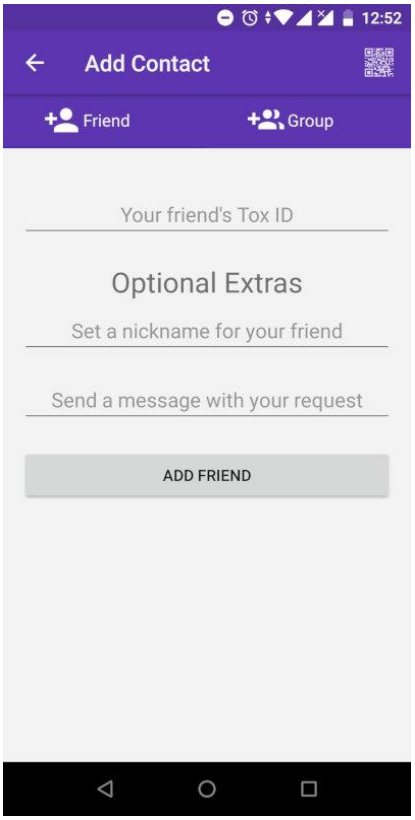


Рисунок 1.11 – Додавання нового контакту в Antox

Розробники пристосували месенджер Antox для різних операційних систем. На десктопних платформах, таких як Windows, macOS і Linux, доступні клієнти qTox і µTox. Antox розроблено спеціально для мобільної платформи Android, а Antidote призначений для користувачів iOS.

Оцінка переваг та недоліків реалізацій месенджера для різних платформ може бути суб'єктивною, оскільки вони можуть залежати від індивідуальних потреб користувачів та особливостей кожної платформи. Однак, загалом, переваги та недоліки можуть включати такі аспекти, як функціональність,

інтерфейс користувача, стабільність, продуктивність, наявність специфічних функцій платформи та сумісність з пристроями.

1.3.5 Клієнт Jami

Синхронізація повідомлень з криптографічним захистом відсутня - повідомлення, що надійшли до облікового запису до моменту входу на новий пристрій, не синхронізуються (рис 1.12).

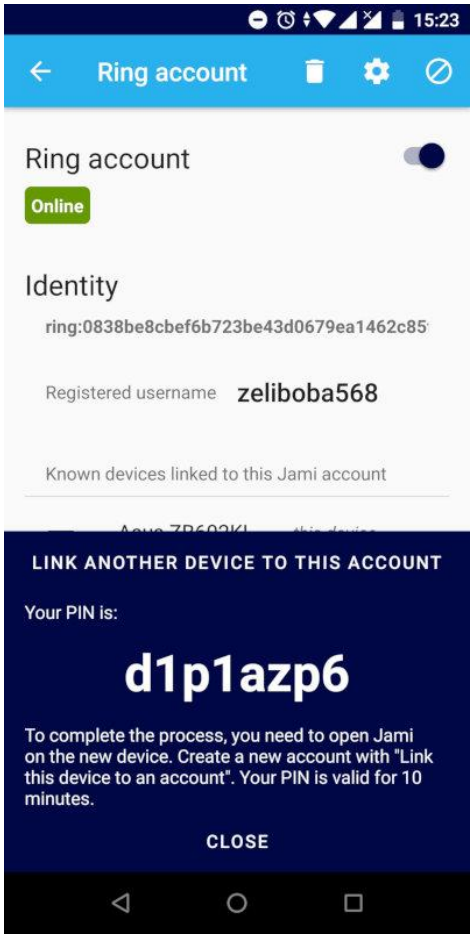


Рисунок 1.12 – Головне вікно Jami

Для додавання контактів в Jami можна сканувати QR-код, але також можна знайти контакт за ніком або ідентифікатором телефону. Ідентифікатор телефону є послідовністю з 40 символів, що служить ідентифікатором користувача. Контактні дані також можна обмінювати через інші месенджери,

електронну пошту або надсилати через Bluetooth. Є можливість додавати кілька облікових записів до одного пристрою за допомогою згенерованого пін-коду.

Jami є повністю відкритим месенджером, код якого публікується під ліцензією GPLv3. Програма э продовженням проекту SFLphone.

Раніше цей месенджер називався Ring, але його перейменували на Jami. Ймовірно, розробники здійснили це, щоб уникнути плутанини з іншим месенджером з такою самою назвою, який не був згаданий у вашому списку.

Як і будь-який інший месенджер, Jami підтримує надсилання текстових повідомлень, безпечні та надійні аудіо- та відеодзвінки, а також передачу документів та файлів.

Для встановлення з'єднання між користувачами використовуються розподілені хеш-таблиці. Усі ключі шифрування та ідентифікації зберігаються на смартфоні користувача. Сервери не використовуються, що означає, що користувачі формують децентралізовану мережу, що сприяє підвищенню безпеки.

1.3.6 Клієнт Chat.Onion

Цей месенджер використовує onion-маршрутизацію(базується на принципі розподіленої мережі, в якій повідомлення пересилаються через кілька проміжних вузлів), яка є основою для браузера Tor. Ця технологія дозволяє приховати IP-адресу користувача, метадані та будь-яку іншу ідентифікуючу інформацію. Використання onion-маршрутизації дозволяє забезпечити високий рівень конфіденційності та анонімності при передачі повідомлень (рис 1.13).

Рисунок 1.13 – Головне вікно Chat.Onion

Для забезпечення анонімності кожне повідомлення проходить через кілька проксі-серверів у випадковому порядку, перш ніж досягає адресата. Це дозволяє зберегти конфіденційність, оскільки кожен сервер знає лише попередній та наступний вузол маршруту.

Кожному користувачеві присвоюється унікальний ідентифікатор (ID) у вигляді 16 символів. Однак, для спрощення процесу, користувач може просто відсканувати QR-код свого співрозмовника, що автоматично встановлює зв'язок між ними.

1.4 Постановка завдання

Розробити веб-сайт для анонімного листування, який дозволить користувачам відправляти повідомлення один одному без розголошування особистої інформації. Сайт повинен забезпечувати безпеку та приватність користувачів, а також забезпечувати надійне шифрування повідомлень.

Сайт повинен мати систему реєстрації, де користувачі можуть створити акаунт за допомогою електронної пошти або іншого ідентифікаційного

методу, який забезпечує їхню анонімність. Необхідно забезпечити безпеку та надійність цієї системи.

Користувачі повинні мати можливість відправляти повідомлення іншим користувачам за допомогою ідентифікатора користувача або електронної пошти. Важливо забезпечити, щоб відправник повідомлення залишався анонімним, іншими словами, отримувач не повинен бачити особистої інформації про відправника.

Для забезпечення конфіденційності повідомлень, вони повинні бути зашифровані на стороні відправника та розшифровані на стороні отримувача. Потрібно застосувати надійні алгоритми шифрування для забезпечення безпеки даних.

Розробити зручний та привабливий інтерфейс, який дозволяє користувачам з легкістю відправляти та отримувати повідомлення. Включіть функції, такі як список повідомлень.

Забезпечити високий рівень анонімності та безпеки на сайті. Захистити особисті дані користувачів, запобігаючи витоку інформації про їхню особистість або інші ідентифікуючі дані. Використовувати надійні методи аутентифікації, шифрування та захисту від несанкціонованого доступу до повідомлень.

Розробити механізми модерування та фільтрації, які допоможуть уникнути неприйнятного вмісту або спаму. Забезпечте можливість користувачам позначати небажані повідомлення, адміністраторам видаляти небажані контент, а також розробіть систему автоматичного виявлення та блокування спаму.

Забезпечити конфіденційність журналів дій користувачів. Зберігайте мінімальну кількість інформації про користувачів та їх дії на сайті. Використовуйте відповідні політики збереження даних та дотримуйтеся регуляторних вимог щодо приватності.

Розробити сайт таким чином, щоб він міг витримувати велику кількість користувачів та навантаження. Забезпечити надійну роботу серверів та швидкий доступ до повідомлень, мінімізуючи можливі перебої та затримки.

Виконати ретельне тестування всіх функцій сайту, включаючи реєстрацію, відправлення повідомлень, шифрування, анонімність та інтерфейс користувача.

1.5 Висновок до розділу

В ході аналізу предметної області, було виявлено основних конкурентів у сфері анонімного спілкування та вивчено їхні пропозиції. Отримані результати дозволяють виявити ринкові тенденції та визначити основні характеристики та конкурентні переваги існуючих рішень.

Враховуючи недоліки існуючих рішень та виявлені потреби користувачів, було розроблено завдання на розробку власного сайту анонімного спілкування.

Результати аналізу предметної області дали нам зрозуміти конкурентне середовище та потреби користувачів. Це дозволить належним чином спроектувати та розробити сайт анонімного спілкування, який відповідає потребам наших користувачів і вирішує проблеми з існуючими рішеннями.

2 ПРОЕКТУВАННЯ ВЕБ РЕСУРСУ ДЛЯ АНОНІМНОГО ЛИСТУВАННЯ

2.1 Концепції створюваного ресурсу

При проектуванні веб ресурсу для анонімного листування було використано технології HLM. HLM (Hidden Lake Messenger) – анонімний месенджер, побудований на ядрі анонімної мережі HLS (Hidden Lake Service).

На сьогоднішній день існує досить велика кількість різноманітних систем спілкування, які тією чи іншою мірою користуються популярністю і мають ті чи інші особливості. Деякі ставлять в основу безпеку зв'язку між користувачами, інші існують виключно заради самого факту збору даних користувача. Тим не менш, вкрай рідко розробляються засоби що дозволяють захищати не саму передану інформацію, а безпосередньо активність користувача - відправлення та отримання інформації [14].

Під приховуванням користувальницької активності ми розумітимемо саме критерій не спостережуваності. Крім критерію не спостереженності, також існує критерій не зв'язуваності, який лише "розриває" зв'язок між абонентами, не дозволяючи зв'язати джерела (відправника) та одержувача. Критерій не зв'язуваності вже присутній у критерії не спостережуваності, а тому є слабшим виразом анонімності.

Критерій не спостережуваності вже включає критерій не зв'язуваності. Якщо піти від зворотного і припустити помилковість даного судження (тобто відсутність не зв'язуваності в не спостережуваності), тоді можна було б за допомогою не зв'язуваності визначити існування суб'єктів інформації і, тим самим, допустити порушення не спостережуваності, що є протиріччям для останнього.

Опис абстрактно-планованого засобу листування приведемо нижче.

"Анонімність" є складним, комплексним терміном. У [5] було представлено розвиток анонімності в мережевих комунікаціях, виявлено критерії анонімності, базове визначення анонімності, основні методи її досягнення (постулати), а також протиріччя між безпекою та анонімністю. Все це говорить про те, що при розробці ми повинні чітко позначити межі того, до чого ми рухаємось.

1. Месенджер буде базуватися на теоретично доведеній анонімності.

Прихованими мережами з теоретично доведеною анонімністю прийнято вважати замкнуті (повністю прослуховуються) системи, в яких стає неможливим здійснення будь-яких пасивних атак (у тому числі і при існуванні глобального спостерігача) спрямованих на деанонімізацію відправника та одержувача з мінімальними умовностями за кількістю вузлів непідпорядкованих. Інакше кажучи, з погляду пасивного атакуючого, апостеріорні знання (отримані внаслідок спостережень) повинні залишатися рівними апріорним (до спостережень), тим самим зберігаючи рівноймовірність деанонімізації по N -му безлічі суб'єктів мережі.

2. Месенджер пов'язуватиме абонентів інформації між собою.

Існує кілька видів анонімізації трафіку між відправником та одержувачем:

1) система розмежовує абонентів інформації. У такій концепції існує три можливі випадки:

- відправник анонімний до одержувача, але одержувач відомий відправнику;

- відправник відомий одержувачу, але одержувач анонімний до відправника;

- відправник та одержувач анонімні один до одного. Прикладом є: анонімний доступ до відкритого Інтернет ресурсу;

2) анонімне отримання інформації з ботнет системи з боку сервера-координатора;

3) анонімний доступ до прихованого ресурсу в анонімній мережі.

2. Система пов'язує абонентів інформації. У такій концепції відправник та одержувач відомі один до одного. Системи побудовані на цьому пункті часто обмежені у своєму застосуванні, але так чи інакше залишаються здатними представляти анонімність суб'єктів, у тому числі і на рівні критерію не спостережуваності.

Другий пункт є найменш привабливим з боку будь-якого правила з першого пункту. Тим не менш, така дія стає виправданою з боку простоти програмної реалізації та простоти теоретичної доказовості.

Месенджер представлятиме наскрізне (end-to-end) шифрування [11].

Клієнт-безпечні програми або програми, що базуються на безпечній лінії зв'язку «клієнт-клієнт», являють собою абстрагування об'єктів, що передаються / зберігаються від проміжних суб'єктів, тим самим наводячи потужність довіри |T| до свого теоретично мінімально заданого значення. Окремим випадком зв'язку "клієнт-клієнт" стає наскрізне (end-to-end або E2E) шифрування.

Потужність довіри — кількість вузлів, що беруть участь у зберіганні або передачі інформації, представленої в них у відкритому описі. Інакше кажучи, такі вузли здатні читати, підміняти і видозмінювати інформацію, для них вона знаходиться у гранично чистому, прозорому, транспарентному стані. Чим більше потужність довіри, тим вище ймовірний шанс компрометації окремих вузлів, а отже, і інформації, що зберігається на них. Прийнято вважати одним із вузлів одержувача. Отже, нульова потужність довіри $|T| = 0$ виникатиме лише у моменти відсутності будь-яких зв'язків і з’єднань. Якщо $|T| = 1$ це говорить про те, що зв'язок захищений, іншими словами, ніхто крім відправника та одержувача інформацією не володіють. В інших випадках $|T| > 1$, Що говорить про груповий зв'язок (тобто, про існування кількох одержувачів), або про проміжні вузли, здатні читати інформацію у відкритому вигляді.

4. Засіб базуватиметься на одноранговій (peer-to-peer) децентралізованій архітектурі мережі.

Існує кілька видів однорангових мереж, а саме:

1) Централізована однорангова архітектура. Є існування двох ролей - клієнтів і ретрансляторів. Клієнт у такій системі генерують та приймають всю інформацію. Ретранслятори служать виключно для перенаправлення клієнтської інформації, не містить ніякої додаткової логіки.

2) Децентралізована однорангова архітектура. Являє собою "зрощування" воєдино клієнтів і ретрансляторів, внаслідок чого кожен клієнт тепер стає здатним здійснювати перенаправлення клієнтської інформації, що надходить йому ззовні.

3) Розподілена однорангова архітектура. Підвид децентралізованої однорангової архітектури. Зародилася внаслідок "корозії" децентралізованих форм централізованими. Іншими словами, у децентралізованих архітектурах існує проблема, коли клієнти починають вибирати малу кількість стабільних клієнтів для подальшої ретрансляції, тим самим призводять систему до неявного централізації. Розподілена однорангова архітектура перекладає якість з'єднань з їхньої кількість.

Вибір децентралізованої архітектури замість розподіленої був викликаний внаслідок п'ятого (f2f мережі) та шостого пунктів (абстрактні анонімні мережі) визначення для побудови засобу анонімного листування. Дані пункти з одного боку обмежують кількість з'єднань, тим самим ускладнюючи будівництво розподіленої системи, з іншого боку централізація (навіть у явному уявленні) не порушуватиме анонімізації трафіку [1].

5. Месенджер буде зв'язувати абонентів через довірчі (friend-to-friend) зв'язки.

Кожен діючий суб'єкт мережі вибудовуватиме зв'язки з іншими учасниками, ґрунтуючись на суб'єктивності до рівня довіри, встановлюючи та редагуючи білий список на своїй стороні. Щоб успішно підключитися до

мережі такого роду, суб'єкту необхідно стати довіреним вузлом, тобто користувачем, якому хтось довіряє [26]. Складність виконання атаки на подібну мережу буде зводитися також до складності вбудовування вузлів, що підпорядковуються, тому що кожен одержувач інформації, зрештою, повинен буде заздалегідь встановлювати список можливих відправників.

6. Засіб листування представлятиме реалізацію абстрактної анонімної мережі. Серед анонімних мереж можна виявити клас систем з теоретично доведеною анонімністю та максимально розмежовує властивістю, що призводить до найбільшого розриву зв'язків між об'єктом (як інформації) та його суб'єктами (в особі відправника та одержувача) за допомогою вибудовування децентралізованих з'єднань між усіма учасниками мережі.

Анонімні мережі, з описаними вище характеристиками, будуть іменуватися абстрактними (рис. 2.1).

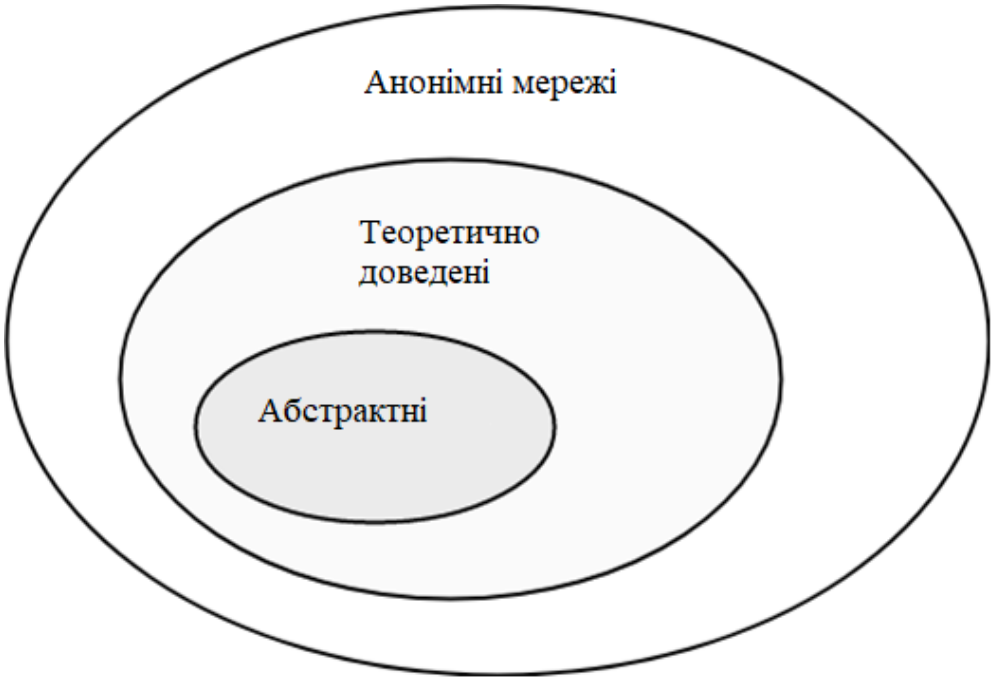


Рисунок 2.1 – Класифікація мереж за ступенем довіри

Через свою специфічну архітектуру передача інформації може здійснюватися в будь-якому дуплексному середовищі незалежно від розташування та зв'язків вузлів, що повністю відриває поширення об'єктів від

своєї мережевої архітектури та переводить маршрутизацію в етап віртуального транслювання.

2.2 Технологія HLS

Коротко суть HLS зводиться до наступного: припустимо, що є три учасника {*A, B, C*}. Кожен з них з'єднаний один до одного (що в порівнянні з DC-мережами не є обов'язковим критерієм, але даний випадок приведено виключно для спрощення) (є реалізації, які дозволяють в DC-мережах не з'єднуватися один до одного, але ці способи швидше є хаками, ніж варіативністю). Кожен суб'єкт встановлює час генерації інформації = *T*. У кожного учасника є своє внутрішнє сховище на кшталт FIFO (перший прийшов – перший пішов), можна сказати структура " черга "[13]. Приведемо можливі схеми зв’язків між вузлами на рисунку 2.2.

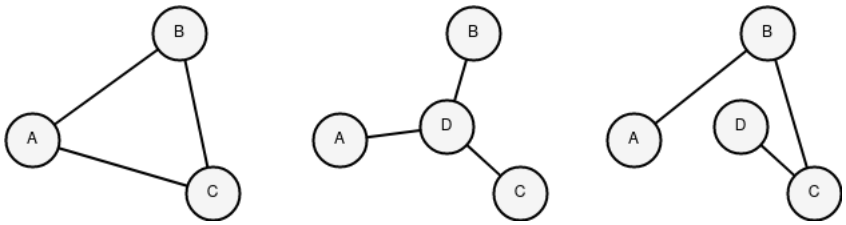


Рисунок 2.2 – Схема зв’язків між учасниками HLS

Припустимо, що учасник *A* хоче надіслати інформацію через мережу так, щоб {*B, C*} цю інформацію отримали, але не змогли дізнатися, хто дійсно є відправником. Іншими словами, для *B* це може бути {*A, C*}, а для C це {*A, B*} з ймовірністю 50/50. Усі учасники починають узгоджувати загальний біт зі своїми сусідами на момент часу *T*. Припустимо, що учасники {*A, B*} узгодили біт = 1, {*B, C*} = 1, {*C, A*} = 0 [7].

Далі кожен учасник мережі виконує операцію XOR (виключаюче АБО) для біт з усіх своїх з'єднань: *A* = 1 xor 0 = 1; *B* = 1 xor 1 = 0; *C* = 0 xor 1 = 1. Дані результати обмінюються по всій мережі та XOR'уються кожним її учасником:

0 xor 1 xor 1 = 0. Це говорить про те, що учасник *A* передав біт інформації = 0. Щоб суб'єкт *A* міг передати біт = 1, йому необхідно додати операцію НЕ у своєму обчисленні, тобто *A* = НЕ(1 xor 0) = 0. У результаті всі обчислення дійдуть такого результату: 0 xor 0 xor 1 = 1. Таким чином, стає можливим передати 1 біт інформації повністю анонімно (звісно ж із боку визначення теоретично доведеної анонімності). Цей процес можна представити схематично, як показано на рисунку 2.3.

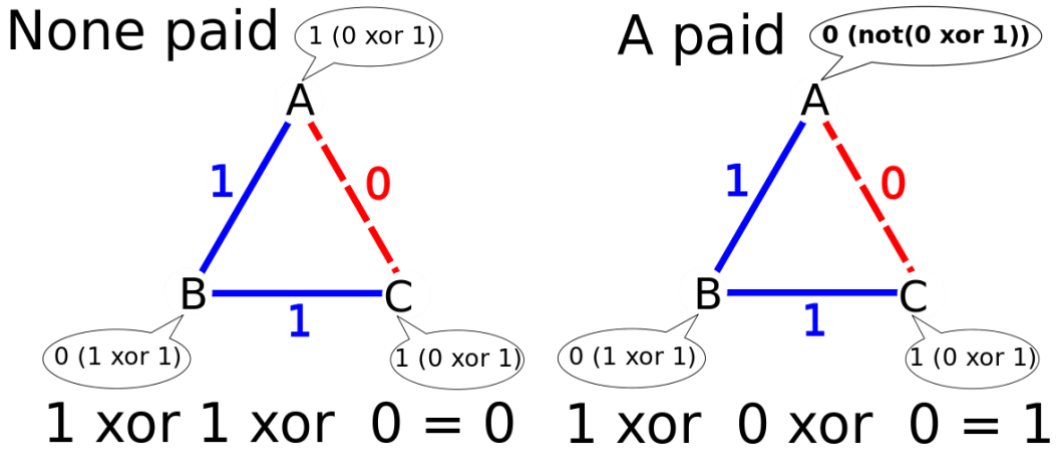


Рисунок 2.3 – Виконання операції XOR учасниками мережі

Припустимо, що учасник *A* хоче надіслати якусь інформацію одному з учасників {*B*, *C*}, так, щоб інший учасник (або зовнішній спостерігач) не знав, що існує якийсь факт відправлення. Кожен учасник у певний період *T* генерує повідомлення. Таке повідомлення може бути або хибним (що не має жодного фактичного змісту і нікому за фактом не відправляється, заповнюючись випадковими бітами), або дійсним (запит чи відповідь). Надіслати раніше або пізніше за визначений час *T* ніякий учасник не може. Якщо зібралося кілька запитів одному й тому учаснику, тоді він їх кладе у свою чергу повідомлень і після періоду *T* дістає з черги і відсилає до мережі. Таким чином, сама структура HLS є множиною послідовно збудованих черг. Схематично це приведено на рисунку 2.4.

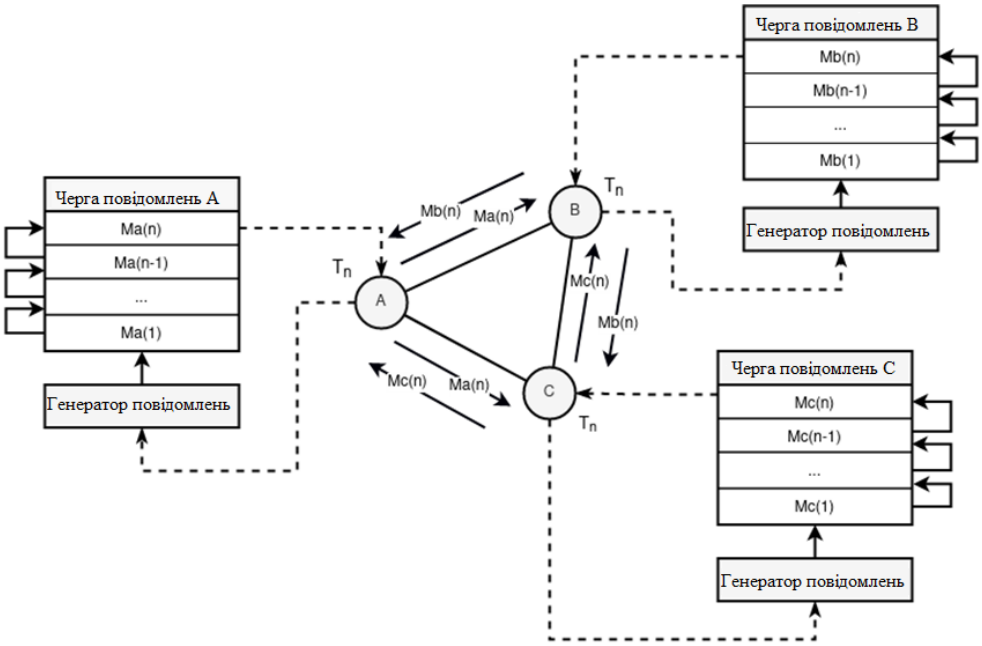


Рисунок 2.4 – Структура HLS

Таким чином, зовнішній глобальний спостерігач бачитиме лише картину, коли він у свій певно заданий період *T* відправляє якесь повідомлення всім іншим вузлам мережі, що не дає жодної інформації про факт відправлення, чи отримання. Внутрішні пасивні учасники також нездатні дізнатися чи комунікує один із учасників у період з будь-яким іншим, так як передбачається, що шифрована інформація не видає жодних даних про відправника та одержувача безпосередньо (як зроблено на приклад в Bitmessage) [3].

Bitmessage – клієнт-безпечний додаток, месенджер. Сам по собі не представляє анонімність користувачів, але його унікальною особливістю є маршрутизація даних, що відправляються.

1. Інформація шифрується (як спрощення) публічним ключем одержувача (насправді використовується гібридна схема шифрування).
2. Зашифрована інформація надсилається всім учасникам мережі (на практиці, звичайно ж, поточним з'єднанням).
3. Кожен користувач при отриманні шифрованої інформації ззовні намагається її розшифрувати своїм приватним ключем.

4. Якщо користувач зміг розшифрувати інформацію, це є справжнім її одержувачем.

5. Якщо користувач не зміг розшифрувати інформацію, він продовжує її розповсюджувати далі по мережі, відправляючи її всім своїм з'єднанням.

Унікальність такого підходу полягає у відсутності інформації, що маршрутизує, крім як розуміння факту {одержувач / не одержувач}. Тим не менш, Bitmessage не є анонімним месенджером, тому що в ньому відсутня будь-яка маршрутизація, що заплутує. Іншими словами, відносно легко визначити, хто є відправником і хто одержувачем, за умови, якщо одержувач завжди буде генерувати відповідь на запит ініціатора.

2.3 Анонімна маршрутизація

Маршрутизація в анонімних мережах не є примітивною і ставить ефективність розповсюдження об'єктів опціональним параметром (низькі / високі затримки) (рис. 2.5), тому що головною метою стає створення алгоритму, що заплутує (анонімізатора), який приводив би до трудомісткості аналізу істинного шляху від точки відправлення до точки призначення [12]. Продуктивність, ефективність "чистої" маршрутизації втрачається, замінюючись особливістю алгоритму. У таких умовах самі приховані мережі стають повільними та складними у застосуванні (у тому числі і з низькими затримками), що також частково чи повноцінно відсуває їх прикладне та повсякденне використання в даний час.

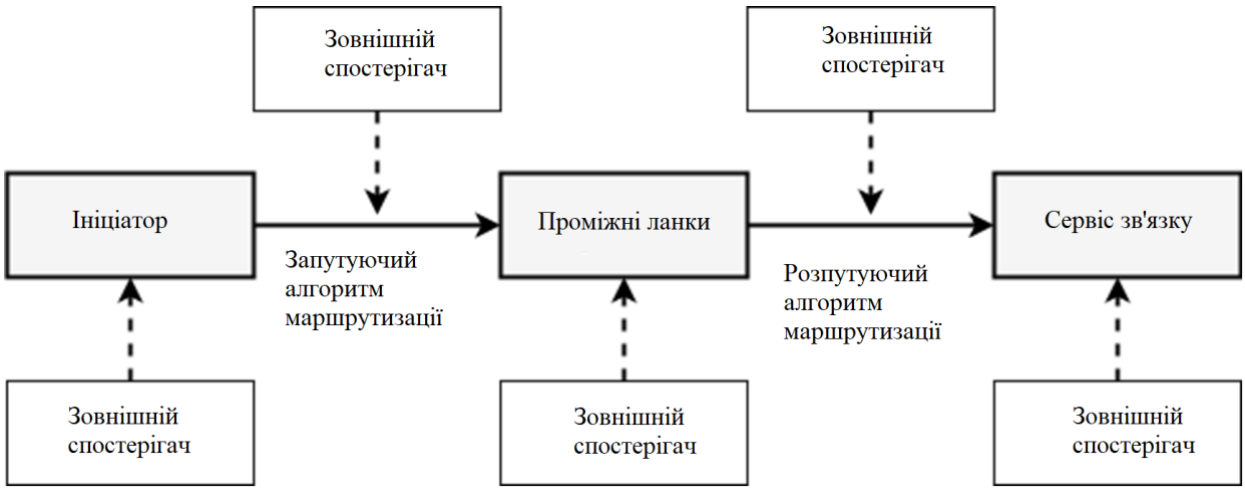


Рисунок 2.5 – Схема анонімної маршрутизації повідомлень

Ядро анонімної мережі має складатися з двох основних складових – криптографічного протоколу та мережевої комунікації. При цьому необхідно не як саме їх поєднання (композиція), а як їх синтез.

Протокол визначається восьма кроками, де три кроки від відправника і п'ять кроків для одержувача. Для роботи протоколу необхідні алгоритми КСГПСЧ (криптографічно стійкого генератора псевдовипадкових чисел), ЕЦП (електронного цифрового підпису), криптографічної хеш-функції, установки/підтвердження роботи, симетричного та асиметричного шифрів.

Учасники протоколу:

A – відправник,

B – одержувач.

Кроки учасника A:

1. K = G(N), R=G(N),

де G – функція-генератор випадкових байт,

N – кількість байт для генерації,

K – сеансовий ключ шифрування,

R – випадковий набір байт.

2. HP = H (R | | P | | PubKA | | PubKB),

де HP – хеш повідомлення,

H – функція хешування,

P – вихідне повідомлення,

PubKX – публічний ключ.

3. CP = [E (PubKB, K), E (K, PubKA), E (K, R), E (K, P), HP, E (K, S (PrivKA, HP)), W (C , HP)],

де CP – зашифроване повідомлення,

E – функція шифрування,

S – функція підписання,

W – функція підтвердження роботи,

C – складність роботи,

PrivKX – приватний ключ.

Схему роботи алгоритму приведено на рисунку 2.6.

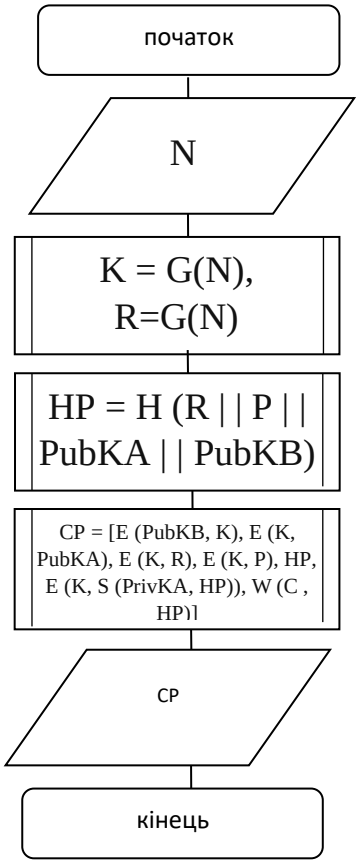


Рисунок 2.6 – Алгоритм відправлення повідомлення

Для учасника B, котрий буде отримувати повідомлення передбачені інші кроки. Схематично вони приведені на рисунку 2.7.

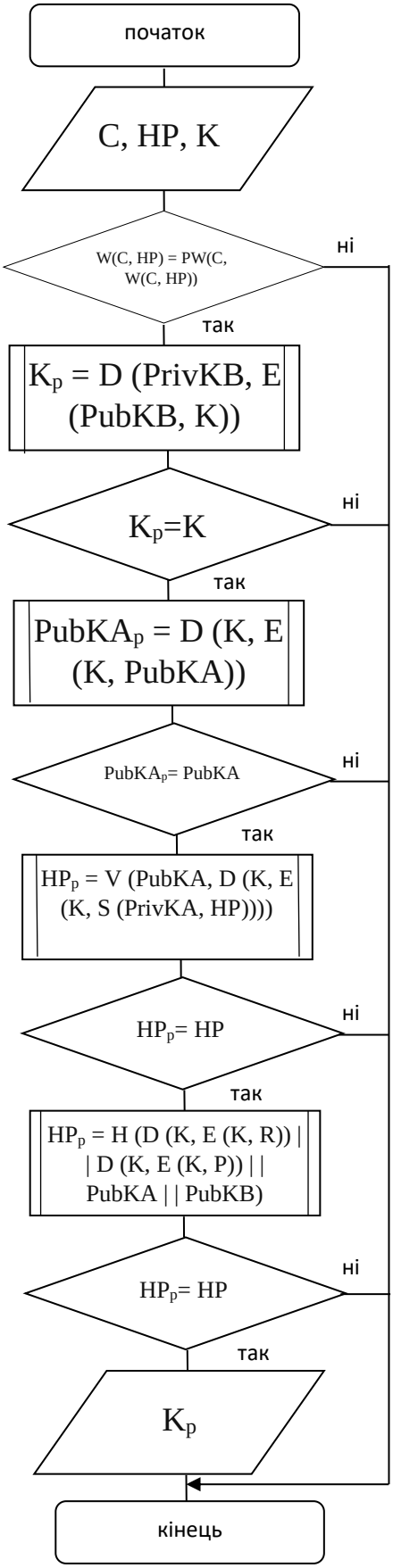


Рисунок 2.7 – Алгоритм на стороні отримувача повідомлення

Кроки учасника B:

4. W(C, HP) = PW(C, W(C, HP)),

де PW – функція перевірки роботи.

Якщо ≠, то протокол переривається.

5. K = D (PrivKB, E (PubKB, K)),

де D – функція розшифрування.

Якщо ≠, то протокол переривається.

6. PubKA = D (K, E (K, PubKA)).

Якщо ≠, то протокол переривається.

7. HP = V (PubKA, D (K, E (K, S (PrivKA, HP)))),

де V – функція перевірки підпису.

Якщо ≠, то протокол переривається.

8. HP = H (D (K, E (K, R)) | | D (K, E (K, P)) | | PubKA | | PubKB),

Якщо ≠, то протокол переривається.

Цей протокол ігнорує спосіб отримання публічного ключа від призначення. Це необхідно через те, щоб протокол був вбудованим і міг впроваджуватися у множину систем, включаючи однорангові мережі, які не мають центрів сертифікації.

Також протокол здатний ігнорувати мережну ідентифікацію суб'єктів інформації, заміщаючи її криптографічною ідентифікацією. При такому підході автентифікація суб'єктів починає ставати сингулярною функцією, що стосується лише і лише асиметричної криптографії, і як наслідок, прикладний рівень стеку TCP/IP починає симулятивно замінювати криптографічний шар за способом виявлення відправника і одержувача. З вищеописаного також справедливо випливає, що для побудови повноцінної інформаційної системи необхідною є симулятивна заміна транспортного та прикладного рівня наступними криптографічними абстракціями. Під транспортним рівнем можна розуміти спосіб передачі повідомлень із зовнішньої (анонімної мережі) у внутрішню (локальну), під прикладним — взаємодію ядра клієнтського коду з внутрішніми сервісами.

Сеанс зв'язку в наведеному протоколі визначається самим пакетом, або іншими словами, один пакет дорівнює одному сеансу за рахунок генерації випадкового сеансового ключа. Описаний підхід призводить до непотрібності збереження фактичного сеансу зв'язку, виключає зовнішні довгострокові зв'язки між суб'єктами у вигляді абстрагування об'єктів, що призводить до неможливості розсекречення всієї інформації, навіть за компрометації одного чи кількох сеансових ключів.

Безпека протоколу визначається переважно безпекою асиметричної функції шифрування, так як всі дії зводяться до розшифрування сеансового ключа приватним ключем [32]. Якщо приватний ключ неспроможний розшифрувати сеансовий, це говорить про те, що саме повідомлення було зашифровано іншим публічним ключем і тому одержувач також є інший суб'єкт.

Функція хешування необхідна для перевірки цілісності відправлених даних. Функція перевірки підпису потрібна для автентифікації відправника. Функція перевірки доказу роботи необхідна для запобігання спаму. Схематично абстрагування криптографічних шарів представлено на рисунку 2.8.

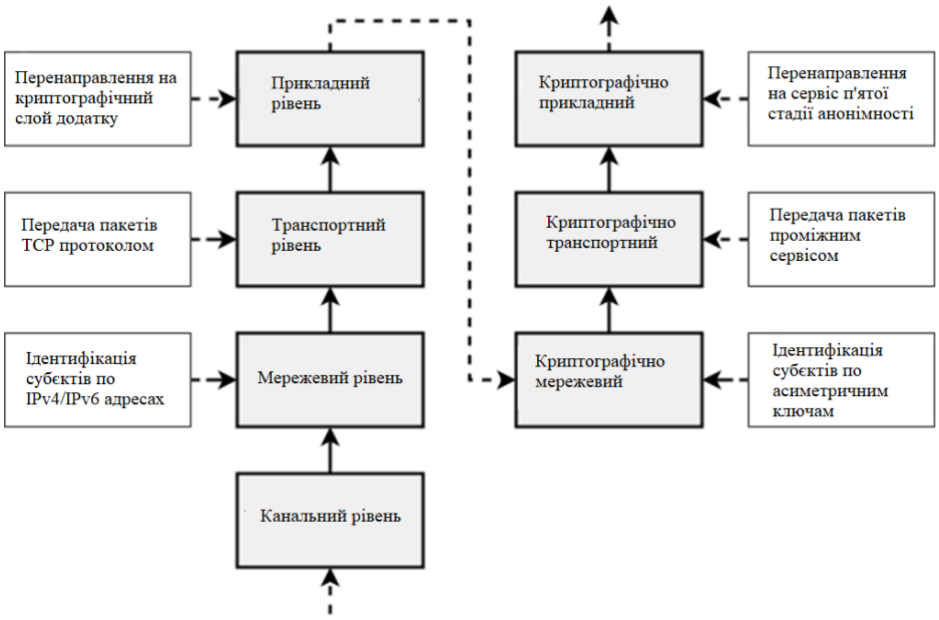


Рисунок 2.8 – Криптографічні шари та їхні зв’язки

Шифрування підпису сеансовим ключем необхідне, так як зловмисник що буде атакувати протокол для визначення відправника (а саме його публічного ключа) може скласти список вже відомих йому публічних ключів і перевіряти кожен на правильність підпису. Якщо перевірка призводить до безпомилкового результату, це говорить про виявлення відправника.

Шифрування випадкового числа також є необхідність, тому що, якщо зловмисник знає його і суб'єктів інформації, що передається, то він здатний пройтися методом «brute force» [21] за словником та поширених текстів для виявлення вихідного повідомлення.

Використання однієї й тієї пари асиметричних ключів для шифрування і підписання перестає бути вразливістю, якщо застосовуються різні алгоритми кодування чи сама структура алгоритму представляє різні способи реалізації. Приміром, при алгоритмі RSA для шифрування може використовуватися алгоритм OAEP, а підписання – PSS [20]. У такому разі не виникає вразливості пов'язаної з можливим чергуванням «шифрування-підписання». Проте залишаються ризики пов'язані з компрометацією єдиної пари ключів, коли зловмисник зможе як розшифровувати всі отримані повідомлення, а й підписувати повідомлення що відправляються. Але цей критерій також є і відносним плюсом, коли особистість суб'єкта не роздвоюється.

Протокол придатний для багатьох завдань, включаючи передачу повідомлень, запитів, файлів, але не придатний для передачі потокової інформації, подібної до аудіо дзвінків та відео трансляцій, через необхідність підписувати та підтверджувати роботу, на що може витрачатись тривала кількість часу. Іншими словами, протокол працює з кінцевою кількістю даних, розмір яких наперед відомий і обробка яких (тобто їх використання) починається з моменту завершення повної перевірки.

Недоліком протоколу є відсутність послідовності між кількома пакетами. Іншими словами неможливо визначити нумерацію, що певною

мірою переводить частину повноцінного протоколу на логіку програми, як, наприклад, передача файлів. Це, у свою чергу, обґрунтовується спрощенням протоколу, де не потрібні сховище або база даних для зберігання послідовності пакетів з боку кожного об'єкта, що входить. Також у деяких додатках послідовність повідомлень не є критичною, як наприклад в електронній пошті або месенджерах, де необхідний лише сам факт вже існуючого дубліката (зараз можна перевіряти хешем пакета).

Іншим недоліком є постійне застосування функції підписання, яка вважається найбільш трудомісткою, з практичної точки зору, операцією. При великій кількості повідомлень, що надходять, виникне і необхідність у великій кількості перевірок підписання. При цьому використання MAC, замість ЕЦП, є неприпустимим, тому що така імітація створить буквально потоковий зв'язок між суб'єктами інформації (створить додаткові зв'язки між суб'єктами і об'єктом, що генерується), ускладнить протокол і може призвести теоретично до більш ніж одного можливого вектора нападу на протокол.

Для покращення ефективності, допустимо при передачі файлів, програмний код можна змінити так, щоб знизити кількість перевірок роботи в процесі передачі, але з початковим доказом роботи на основі випадкового рядка (отриманого від точки призначення), а потім і з накопиченим хеш-значенням з n -Блоків файлу, для i -ої перевірки. Таким чином, мінімальний контроль роботи буде здійснюватися лише $M/nN+1$ раз, де M розмір файлу, N розмір одного блоку. Якщо доказ не надійшов або він є невірним, потрібно вважати, що файл був переданий з помилкою і тим самим запитати пошкоджений або неперевірений блок заново.

Інша проблема полягає у відсутності будь-яких видимих метаданих (хеш-значення, докази роботи), які б допомогли у боротьбі зі спамом, що у свою чергу є вкрай важливим критерієм для більшості децентралізованих систем. Таким чином, відсутність метаданих рівносильна відсутності відмовостійкості, що відсилає на суперечність еквівалентності повністю аналізованого та не схильного аналізу пакетів. Одним із можливих рішень цієї

проблеми може бути використання загальноприйнятного та стандартизованого протоколу типу SSL/TLS з метою приховування факту використання монолітного протоколу.

2.4 Реалізація основних компонентів системи

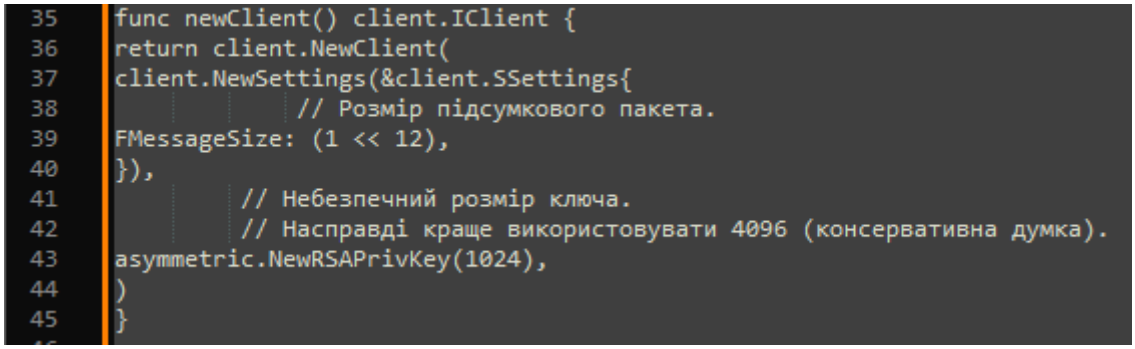
Програмна реалізація шифрування та розшифрування наведена на рисунку 2.9. Основною відмінністю програмної реалізації від концептуально описаного протоколу є облік статики пакета. Іншими словами, потрібна така властивість, щоб один пакет був невідмінним від іншого за своїм розміром.

Приклад використання шифрування та розшифрування можна продемонструвати в такий спосіб, як це показано на рисунку 2.9.

```
11  func main() {
12      var (
13          client1 = newClient()
14          client2 = newClient()
15      )
16
17      msg, err := client1.Encrypt(
18          client2.PubKey(),
19          payload.NewPayload(0x0, []byte("hello, world!")),
20      )
21      if err != nil {
22          panic(err)
23      }
24
25      pubKey, pld, err := client2.Decrypt(msg)
26      if err != nil {
27          panic(err)
28      }
29
30      fmt.Printf("Message: '%s';\nSender's public key: '%s';\n", string(pld.Body()), pubKey.String())
31      fmt.Printf("Encrypted message: '%s'\n", string(msg.ToBytes()))
32  }
33
```

Рисунок 2.9 – Реалізація функції шифрування

В головній функції створюється два клієнти. Перший клієнт шифрує повідомлення та створює пакет для відправки. При реалізації тестового випадку було використано асиметричне шифрування з ключем 1024 біти [28]. Частина коду що відповідає за розмір ключа та перевірку розміру пакету приведена на рисунку 2.10.



```
func newClient() client.IClient {
return client.NewClient(
client.NewSettings(&client.SSettings{
                // Розмір підсумкового пакета.
FMessageSize: (1 << 12),
}),
            // Небезпечний розмір ключа.
            // Насправді краще використовувати 4096 (консервативна думка).
asymmetric.NewRSAPrivKey(1024),
)
}
```

Рисунок 2.10 – Встановлення розміру ключа та перевірка підсумкового пакету

Результат виконання функції шифрування та параметрів що використовувались приведено на рисунку 2.11.

```
Message: 'hello, world!';
Sender's public key: 'Pub(go-peer/rsa){30818902818100CC228131C038583D6345EEFF79D5A6AD56EB3992CD1933655EC1830F66AAF8F9CAC72
A7C2E3905DD28466F57F3FA0F53F0EF724D109D0B120CD9CF49DAF4841EE22F86EBD6A498DF91518C52C78583E7D61509C5E37906 93D16625432185E1677F3F0BD50203010001}';
Encrypted message: '{"head":{"salt":"86232459f3a46d6f8bb2d45d1d39898d047066b92606ff125d5c09b484c56558518c2c6b08599 4cd357031a257c2f8f34631119",
"session":"c32d0e7e1b731f802de593104e5b062cfbda16437f89ede1c0b33a75fb008bd22
2d00adb2fd18ae4521c77f37da9199dad763b8a159caef9b5965527ede8b4ec4f43e16388845f41d07418b5abf3af22cad25cb546c5a
1b07f4fe1bb0ddb108a1d51de","sender":"2b66ea223b48ae813b62e8a50eb059839f7b7f661ab67c9cebed54fe62a701a341b7bce789
012c4586a3c31048085d5325db90f8c87794be38ad6b8fdc81cb2f983ead4fb57e5e231a41e740d6540c1f3f0e2012282992
7cb744549d31792a260a7d6ea34805f15fc35978784f431ccd4a030b1f3b43ddd2d3fe681053c9584596f13"},"body":
{"payload":"2cc55c74ecbd7fa0f75e2ce97678cd86e7920a8c919e063e43525216e18c53cf8918b0a18c3b93d57607ac1ec9e8e4f43f6df12d31bf15db56202ab63fad8e1
d898a7bf9e9add90cef4d84e7e2c2cc07ac13ee823efdc4397c5c539f8ee1e32b37e851399f7f683a58b7a64e3873c6c84b9738dbab35
12d05ba55e96e133926a69e6af68d8c29123802058e75db1726d164b245b1afbb4d2bdb11884f837f5a642d72260a2f660d794a2ebbc044a74662676e5b20b
3ae24c32c0074d330fb55b895bd08dddbd1b04cf88d13a9fbc062312d859156f1a36967cd4f01f8c794b7844ab61c7e66ea83bb0f25595c78750131434fc024869e85eb3
eb8e51fde56fa8d49df5ab41c41197712c1b38af8f31c4063626e1326e129e1ac1cb53570eb820fc2d0b6dceff3f622f88921f64405f5dbefb2452652902e
d1bc211495f19a5cabe1c71cc3334ae8290feaec9018e00074ce7298ff8fc09f815eb839f2f95ce78d8f1e7dc4be785de62b616b3e7061288b07b27
29f5813a3382f1ca0a5f8821436a36a94b2447d86a6d2356c61981aa095eb56df8cddae9cb6060a069d45301085a8bdea635a770f8c8a082560f40",\
"sign":"14438df8beb9f31a15fef4acbc644e5fd6401ca83a7e1154039564215b4682bac6cfc577c0174227c4400c419b212a02857736
8ced28bc6feb14924bd9d44650ae9abd67940dde8c6aae390015e67cca723c0e574da75006d0b15f2b225d44696da834277f5c4a5
b0013e7ee7baedfb2b00a20f8ea78848d8b49563fc32d24dfd8550436c2","hash":"d8873b265353fd564a95c6b122fa8783499 97b","proof":"0000000000001419"}}'
```

Рисунок 2.11 – Зашифрований пакет

Мережева складова ядра прихованої мережі базуватиметься на протоколі TCP. Необхідність TCP над UDP полягає в розумінні точної доставки всіх пакетів від точки A до точки B [6]. Необхідність TCP над HTTP полягає в постійному триманні з'єднання між відправником і одержувачем. У будь-якому випадку можна замінити TCP, модифікувавши UDP протокол, або використавши вебсокети на рівні HTTP, але це лише ускладнюватиме систему без будь-якої значної позитивної сторони.

Потрібно визначити базові функції надсилання інформації. Таких буде дві. Broadcast є функцією поширення інформації по всіх поточних з'єднаннях і тому прив'язаний до об'єкта "зберігача" з'єднань (у нашому контексті –

це вузол). Request є функцією передачі до одного з'єднанню з метою отримати від нього відповідь і тому прив'язаний до об'єкта " з'єднання ".

Функція Broadcast забезпечує збереження хешу в пам’яті, щоб запобігти безкінечному циклу відправки пакета та готує корисне навантаження. Етап розробки функції приведено на рисунку 2.12.

```
func (node *sNode) Broadcast(pld payload.IPayload) error {
// Збереження хешу у пам'яті, щоб запобігти нескінченному
     // зациклювання пакета мережевої передачі
     hash := hashing.NewSHA256Hasher(pld.ToBytes()).Bytes()
node.inMappingWithSet(hash)

     // Беремо всі поточні з'єднання та відправляємо кожному
     // Вузлу копію корисного навантаження.
var err error
for _, conn := range node.Connections() {
e := conn.Write(pld)
if e != nil {
err = e
}
}

return err
}
```

Рисунок 2.12 – Реалізація функції Broadcast

Це звичайно метод, а не функція з боку термінології мови Go. Тим не менш, термін "функція" вживається тут і далі як певна дія, як деяка процедура без прив'язки безпосередньо до мови програмування.

Функція Request дозволяє відправити корисне навантаження та отримати відповідь, щоб підтвердити, що дані валідні. Реалізація функції приведена на рисунку 2.13.

```
func (conn * sConn) Request (pld payload.IPayload) (payload.IPayload, error) {
var (
chPld = make(chan payload.IPayload)
timeWait = conn.fSettings.GetTimeWait()
)

     // Відправляємо корисне навантаження.
if err := conn.Write(pld); err != nil {
return nil, err
}
     // Запускаємо горутину і намагаємось отримати відповідь.
go readPayload(conn, chPld)

select {
case rpld := <-chPld:
          // Прийняті дані можуть виявитися невалідними.
if rpld == nil {
return nil, fmt.Errorf("failed: read payload")
}
return rpld, nil
case <-time.After(timeWait):
return nil, fmt.Errorf("failed: time out")
}
}
```

Рисунок 2.13 – Реалізація функції Request

Для перевірки на валідність корисного навантаження створена функція readPayload, результатом якої буде висновок про коректність. Реалізація функції приведена на рисунку 2.14.

```
93  func readPayload(conn *sConn, chPld chan payload.IPayload) {
94  //Результатом функції стане висновок корисного навантаження.
95      var pld payload.IPayload
96  defer func() {
97  chPld <- pld
98  }()
99
```

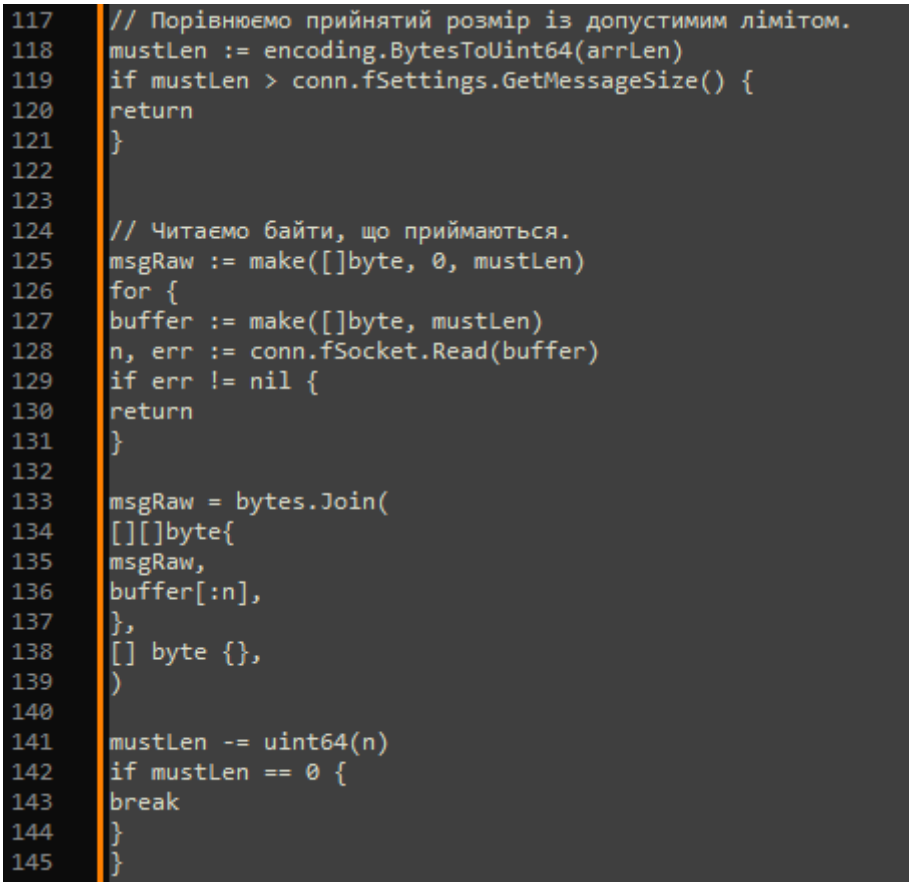
Рисунок 2.14 – Реалізація функції readPayload

Код функції забезпечує читання даних та їх перекодування. Також виконується копіювання в новий масив. Реалізація частини коду приведена на рисунку 2.15.

```
101 // Намагаємося прочитати блок у 64біт вказівний
102 // Розмір даних, що приймаються.
103 bufLen := make([]byte, encoding.CSizeUint64)
104 length, err := conn.fSocket.Read(bufLen)
105 if err != nil {
106 return
107 }
108 if length != encoding.CSizeUint64 {
109 return
110 }
111
112 // mustLen = Size[u64] in uint64
113 arrLen := [encoding.CSizeUint64]byte{}
114 copy(arrLen[:], bufLen)
```

Рисунок 2.15 – Частина функції readPayload

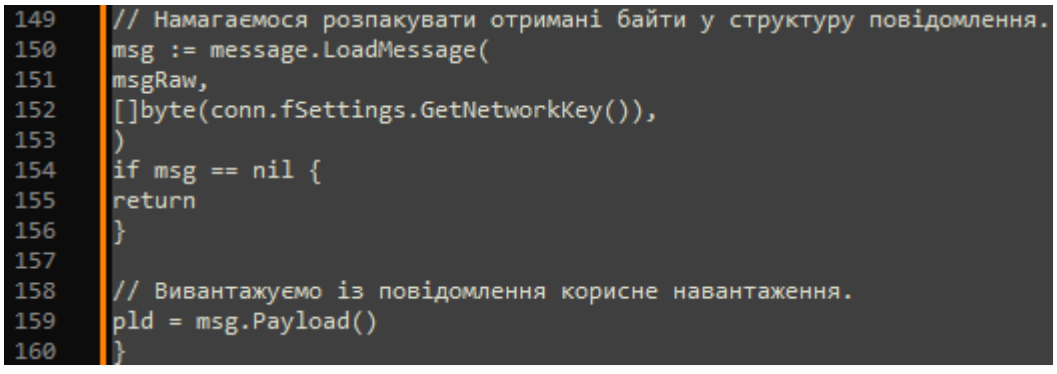
Далі виконується порівняння розміру буферу з допустимим лімітом та копіювання даних з буфера сокету в масив з яким будемо працювати. Також є необхідним перекодування в схему UINT64. Реалізація цієї частини коду приведена на рисунку 2.16.



```
// Порівнюємо прийнятий розмір із допустимим лімітом.
mustLen := encoding.BytesToUint64(arrLen)
if mustLen > conn.fSettings.GetMessageSize() {
return
}


// Читаємо байти, що приймаються.
msgRaw := make([]byte, 0, mustLen)
for {
buffer := make([]byte, mustLen)
n, err := conn.fSocket.Read(buffer)
if err != nil {
return
}

msgRaw = bytes.Join(
[][]byte{
msgRaw,
buffer[:n],
},
[] byte {},
)

mustLen -= uint64(n)
if mustLen == 0 {
break
}
}
```

Рисунок 2.16 – Частина функції readPayload

Код для розпакування отриманих даних в структуру з повідомленням та вивантаження із неї корисного навантаження приведено на рисунку 2.17.



```
// Намагаємося розпакувати отримані байти у структуру повідомлення.
msg := message.LoadMessage(
msgRaw,
[]byte(conn.fSettings.GetNetworkKey()),
)
if msg == nil {
return
}

// Вивантажуємо із повідомлення корисне навантаження.
pld = msg.Payload()
}
```

Рисунок 2.17 – Частина функції readPayload

У функції readPayload використовується згадка Message, проте це інша структура, відмінна від Message у криптографічному протоколі. Загальна назва була взята через непряме, непряме застосування в момент

передачі. Іншими словами, в мережевих комунікаціях передається не саме корисне навантаження, а корисне навантаження запаковане в структуру Message. Рівно за таким же сценарієм у криптографічному протоколі передається не саме корисне навантаження, а структура Message, в якій міститься зашифроване корисне навантаження.

Далі визначимо функцію прийняття з'єднання та всієї наступної інформації від даного з'єднання.

Функція Handle приведена на рисунку 2.18. Функція містить перемикач що надає їй можливість роутингу повідомлень.

```
168 // Аргументи = вузол приймає інформацію, з'єднання відправника,
169 // Відправлене корисне навантаження.
170 type IHandlerF func(INode, conn.IConn, payload.IPayload)
171
172 func (node *sNode) Handle(head uint64, handle IHandlerF) INode {
173 node.fMutex.Lock()
174 defer node.fMutex.Unlock()
175
176     // Встановлюємо функцію як роутинг.
177 node.fHandleRoutes[head] = handle
178 return node
179 }
180 func (node *sNode) handleConn(address string, conn conn.IConn) {
181 defer node.Disconnect(address)
182 for {
183         // Якщо повідомлення прийнято валідне, тоді намагаємось
184         // Прочитати ще одне повідомлення від цього вузла.
185 ok := node.handleMessage(conn, conn.Read())
186 if !ok {
187             // Інакше обриваємо з ним з'єднання.
188 break
189 }
190 }
191 }
```

Рисунок 2.18 – Реалізація функції Handle

Аргументами функції будуть: вузол, та з’єднання відправника. В ній вбудована перевірка на валідність повідомлення. Якщо прочитане повідомлення валідне то функція спробує прочитати наступні повідомлення від відправника, якщо ж ні з’єднання буде розірвано. Частиною функції Handle є використання функції handleMessage, реалізація якої приведена на рисунку 2.19.

```
193 func (node *sNode) handleMessage(conn conn.IConn, pld payload.IPayload) bool {
194 // Перевіряємо валідність прийнятого корисного навантаження
195 if pld == nil {
196 return false
197 }
198
199 // Перевіряємо повідомлення в мапінгу (чи приймало воно раніше?)
200 hash := hashing.NewSHA256Hasher(pld.ToBytes()).Bytes()
201 if node.inMappingWithSet(hash) {
202         // Це не є помилкою, тому що відправник
203         // може отримати пакет із різнорідних вузлів.
204 return true
205 }
```

Рисунок 2.19 – Реалізація функції handleMessage

Функція handleMessage виконує опрацювання отриманого пакету, перевіряє його валідність та чи було воно отримане раніше за допомогою мапінгу. Частина коду функції, що отримує корисне навантаження приведена на рисунку 2.20.

```
207 // Намагаємося отримати функцію роуту. Якщо такої немає
208     // або вона невизначена, тоді вважаємо, що це помилка
209     // за відправника.
210 f, ok := node.getFunction(pld.Head())
211 if !ok || f == nil {
212 return false
213 }
214
215     // Обробляємо корисне навантаження одержаною функцією.
216 f(node, conn, pld)
217 return true
218 }
219 func (conn *sConn) Read() payload.IPayload {
220 chPld := make(chan payload.IPayload)
221 go readPayload(conn, chPld)
222 return <-chPld
223 }
```

Рисунок 2.20 – Частина коду функції handleMessage, що отримує корисне навантаження

Функція також намагається маршрут від кого було отримане повідомлення, якщо такого немає то повідомлення вважається не валідним.

2.5 Реалізація мережевої взаємодії кількох клієнтів

Приклад взаємодії кількох користувачів можна показати в нескінченному циклі передачі інформації від ініціатора до відправника і навпаки. Реалізуємо функцію, що буде забезпечувати постійну взаємодію двох клієнтів, для перевірки розроблених функцій системи. На рисунку 2.21 приведена частина коду, що забезпечує створення двох мережевих сервісів, котрі будуть взаємодіяти.

```
228 package main
229
230 import (
231     "fmt"
232     "strconv"
233     "time"
234
235     "/go-peer/modules/network"
236     "/go-peer/modules/network/conn"
237     "/go-peer/modules/payload"
238 )
239
240 const (
241     serviceHeader  = 0xDEADBEAF
242     serviceAddress = ":8080"
243 )
244
245 func main() {
246     var (
247         service1 = network.NewNode(network.NewSettings(&network.SSettings{}))
248         service2 = network.NewNode(network.NewSettings(&network.SSettings{}))
249     )
250
251     service1.Handle(serviceHeader, handler("#1"))
252     service2.Handle(serviceHeader, handler("#2"))
253
254     go service1.Listen(serviceAddress)
255     time.Sleep(time.Second) // wait
256
257     _, err := service2.Connect(serviceAddress)
258     if err != nil {
259         panic(err)
260     }
261
262     //
263     //
264     service2.Broadcast(payload.NewPayload(
265         serviceHeader,
266         []byte("0"),
267     ))
268
269     select {}
270 }
```

Рисунок 2.21 – Функція, що реалізовує взаємодію 2 клієнтів

Функція надає можливість перевірити маршрутизацію повідомлень та роботу розроблених модулів читання та відправки повідомлень. Частина коду з відправкою повідомлення приведена на рисунку 2.22.

```
272 func handler(serviceName string) network.IHandlerF {
273     return func(n network.INode, c conn.IConn, p payload.IPayload) {
274         time.Sleep(time.Second)
275
276
277         num, err := strconv.Atoi(string(p.Body()))
278         if err != nil {
279             panic(err)
280         }
281
282         val := "ping"
283         if num%2 == 1 {
284             val = "pong"
285         }
286
287         fmt.Printf("service '%s' got '%s#%d'\n", serviceName, val, num)
288
289
290         n.Broadcast(payload.NewPayload(
291             serviceHeader,
292             []byte(fmt.Sprintf("%d", num+1)),
293         ))
294     }
295 }
```

Рисунок 2.22 – Частина коду для мережевої взаємодії

Результат виконання тестової функції буде безкінечне спілкування клієнтів в мережі, як це показано на рисунку 2.23.

```
service '#1' got 'ping#0'
service '#2' got 'pong#1'
service '#1' got 'ping#2'
service '#2' got 'pong#3'
service '#1' got 'ping#4'
service '#2' got 'pong#5'
service '#1' got 'ping#6'
service '#2' got 'pong#7'
service '#1' got 'ping#8'
service '#2' got 'pong#9'
service '#1' got 'ping#10'
service '#2' got 'pong#11'
...
```

Рисунок 2.23 – Результати мережевої взаємодії двох сервісів

Виконаємо фінальний етап концепції HLS, а саме об'єднання мережевої

комунікації та криптографічного протоколу. Сам криптографічний протокол досить легко абстрагується від мережевих комунікацій, такий здатний самостійно і симулятивно замінювати мережеву комунікацію (ідентифікацію) криптографічною. Іншими словами, всю інформацію ми транспортуватимемо і маршрутизуватимемо не на базі IP-адрес, а на основі публічних ключів. Самі публічні ключі стануть мережними ідентифікаторами.

Перше, що необхідно реалізувати – це черга повідомлень (криптографічних). У такій структурі повинен бути закладений механізм генерації випадкових (хибних) повідомлень, щоб постійно і потоково приховувати будь-який факт відправлення або отримання інформації.

Для заповнення черги створена функція Enqueue. Вона здійснює контроль процесу розміщення та перевірку наповненості черги. Етап розробки функції приведено на рисунку 2.24.

```
317 func (q *sQueue) Enqueue(msg message.IMessage) error {
318 q.fMutex.Lock()
319 defer q.fMutex.Unlock()
320
321     // Якщо черга переповнена -> видавати помилку.
322 if uint64(len(q.fQueue)) >= q.Settings().GetCapacity() {
323 return errors.New("queue already full, need wait and retry")
324 }
325
326     // Розмістити повідомлення у чергу.
327 go func() {
328 q.fMutex.Lock()
329 defer q.fMutex.Unlock()
330
331 q.fQueue <- msg
332 }()
333
334 return nil
335 }
```

Рисунок 2.24 – Реалізація функції Enqueue

Для отримання повідомлень з черги створено функцію Dequeue. Функція здійснює перевірку на активність та якщо немає повідомлень то генерує повідомлення про те що черга пуста. Реалізація функції приведена на рисунку 2.25.

```
317 func (q *sQueue) Dequeue() <-chan message.IMessage {
318 time.Sleep(q.Settings().GetDuration())
319
320 go func() {
321 q.fMutex.Lock()
322 defer q.fMutex.Unlock()
323
324             // Якщо черга неактивна -> зупинити виконання.
325 if !q.fIsRun {
326 return
327 }
328
329             // Якщо у черзі повідомлень немає повідомлень,
330             // Тоді потрібно взяти хибне повідомлення з пулу згенерованих.
331 if len(q.fQueue) == 0 {
332 q.fQueue <- (<-q.fMsgPull.fQueue)
333 }
334 }()
335
336 return q.fQueue
337 }
```

Рисунок 2.25 – Реалізація функції Dequeue

Далі необхідним є створення обгортки над функціями відправлення та прийняття повідомлень у мережевих комунікаціях, але змінивши при цьому ідентифікатори. Іншими словами, необхідно замінити одержувача network.INode на anonymity.INode та відправника conn.IConn на asymmetric.IPubKey.

Реалізація функції Handle приведена на рисунку 2.26.

```
342 type IHandlerF func(INode, asymmetric.IPubKey, payload.IPayload) []byte
343
344 func (node *sNode) Handle(head uint32, handle IHandlerF) INode {
345     node.fMutex.Lock()
346     defer node.fMutex.Unlock()
347
348     node.fHandleRoutes[head] = handle
349     return node
350 }
```

Рисунок 2.26 – Реалізація функції Handle

Варто зауважити, що head раніше дорівнював 64bit, тепер 32bit. Пов'язано це з тим, що в "обертковій" реалізації нам також необхідно створити механізм отримання відповіді. Дана реалізація більш обмежена, ніж

Request у мережевих комунікаціях, тому що невідомо наскільки далеко і в якій частині мережі знаходиться кінцевий адресат повідомлення. У такій парадигмі інформація проходитиме кілька вузлів і відповідь одержувача може бути прийнята не тим вузлом, через який спочатку надсилалося повідомлення.

Тому тут 64bit розбивається на дві складові по 32bit як конкатенації. Ліва половина відповідає за унікальний ідентифікатор зв'язку між відправником і одержувачем для правильного валідування інформації, що приймається, і перенаправлення такої на функцію Request як відповідь. Права половина відповідає за функції редиректу, як і в мережевих комунікаціях 64bit. Реалізація коду функції маршрутизаці приведена на рисунку 2.27.

```
355  // Єдина роут функція із мережевих комунікацій.
356  // Є обгорткою над безліччю роут функцій
357  / / Анонімних комунікацій.
358  func (node *sNode) handleWrapper() network.IHandlerF {
359  go func() {
360  for {
361                  // Якщо з черги надійшло повідомлення,
362                  // його необхідно відправити в мережу.
363  msg, ok := <-node.Queue().Dequeue()
364  if !ok {
365  break
366  }
367  node.broadcast(msg)
368  }
369  }()
370
371  return func(nnode network.INode, _ conn.IConn, npld payload.IPayload) {
372  // Отримуємо повідомлення із мережі. Перевіряємо. Розпаковуємо.
373          msg := node.initialCheck(message.LoadMessage(npld.Body()))
374  if msg == nil {
375  return
376  }
```

Рисунок 2.27 – Реалізація процесу маршрутизації

Залишилося лише єдина деталь у тому, щоб ядро прихованої мережі могло називатися справді ядром – надання API стороннім додаткам.

1. GET/POST/DELETE /api/config/connects
2. GET/POST/DELETE /api/config/friends
3. GET/DELETE /api/network/online
4. POST/PUT /api/network/push
5. GET /api/node/pubkey

Як приклад підсумкового результату можна звернутися до директорії examples/cmd/echo_service. Розгортається кілька вузлів, саме – один відправник, один одержувач (сервіс) і проміжний вузол (створений винятково для маршрутизації. Процес розгортання тестових вузлів приведений на рисунках 2.28 – 2.29.

```
$ cd examples/cmd/echo_service
$ make
$ ./request.sh
> HTTP/1.1 200 OK
> Content-Type: application/json
> Date: Fri, 03 Jun 2023 08:02:51 GMT
> Content-Length: 97

> {"result":"7b226563686f223a2268656c6c6f2c20776f726c6421222c2272657475726e223a317d0a","return":1}
```

Рисунок 2.28 – Результат роботи скрипта з запитами

Сам скрипт request.sh містить JSON параметри та формат запиту.

```
#!/bin/bash

str2hex() {
    local str=${1:-""}
    local fmt="%02X"
    local chr
    local -i i
    for i in `seq 0 $((${#str}-1))`; do
        chr=${str:i:1}
        printf "${fmt}" "'${chr}"
    done
}

JSON_DATA='{
        "method":"POST",
        "host":"hidden-echo-service",
        "path":"/echo",
        "head":{
            "Accept": "application/json"
        },
        "body":"aGVsbG8sIHdvcmxkIQ=="
}';

PUSH_FORMAT="{
        \"receiver\":\"Pub(go-peer/rsa){3082020A0282020100B752D35E81F4AEEC1A9C42EDED16E8924DD4D359663611DE2D0
        \"hex_data\":\"$(str2hex "$JSON_DATA")\"
}";

curl -i -X POST -H 'Accept: application/json' http://localhost:7572/api/network/push --data "${PUSH_FORMAT}"
```

Рисунок 2.29 – Скрипт request.sh

У цьому контексті ми скористалися HLS для відправлення запиту з метою отримати відповідь від сервісу, розташованого за нодою, публічний ключ якої = Pub(gopeer/rsa){3082020A1282020101B752D35E81F4... 8560C3F31CC702130.

Ми відправили "hello, world". Отримали hex-кодування. Якщо розкодуємо, то отримаємо таку відповідь: `{"echo":"hello, world!", "return":1}`. HLS повертає відповідь (від функції Request) завжди в hex кодуванні за успішного return'e = 1.

Сам сервіс для відправки тестових повідомлень був написаний в такий спосіб, як приведено на рисунку 2.30.

```
762  package main
763
764  import (
765      "encoding/json"
766      "io"
767      "net/http"
768  )
769
770  type sResponse struct {
771      FEcho   string `json:"echo"`
772      FReturn int    `json:"return"`
773  }
774
775  func main() {
776      http.HandleFunc("/echo", echoPage)
777      http.ListenAndServe(":8080", nil)
778  }
779
780  func echoPage(w http.ResponseWriter, r *http.Request) {
781      if r.Method != "POST" {
782          response(w, 2, "failed: incorrect method")
783          return
784      }
785      res, err := io.ReadAll(r.Body)
786      if err != nil {
787          response(w, 3, "failed: read body")
788          return
789      }
790      response(w, 1, string(res))
791  }
792
793  func response(w http.ResponseWriter, ret int, res string) {
794      w.Header().Set("Content-Type", "application/json")
795      json.NewEncoder(w).Encode(&sResponse{
796          FEcho:   res,
797          FReturn: ret,
798      })
799  }
```

Рисунок 2.30 – Приклад сервісу (echo)

Перенаправлення HLS на послуги реалізовано досить легко. Код що забезпечує процес пере направлення приведений на рисунку 2.31.

```
package handler

import (
"bytes"
"fmt"
"io"
"net/http"

"/go-peer/cmd/hls/config"
hls_network "/go-peer/cmd/hls/network"
hls_settings "/go-peer/cmd/hls/settings"
"/go-peer/modules/crypto/asymmetric"
"/go-peer/modules/network/anonymity"
"/go-peer/modules/payload"
)

func HandleServiceTCP(cfg config.IConfig) anonymity.IHandlerF {
return func(node anonymity.INode, sender asymmetric.IPubKey, pld payload.IPayload) []byte {
// Отримуємо запит із корисного навантаження.
requestBytes := pld.Body()
request := hls_network.LoadRequest(requestBytes)
if request == nil {
return nil
}

// Чи бачимо такий хост у наших сервісах.
address, ok := cfg.Service(request.Host())
if !ok {
return nil
}
```

Рисунок 2.31 – Перенаправлення HLS

Коротко структуру розробленого HLS можна зобразити так, як приведено на рисунку 2.32.

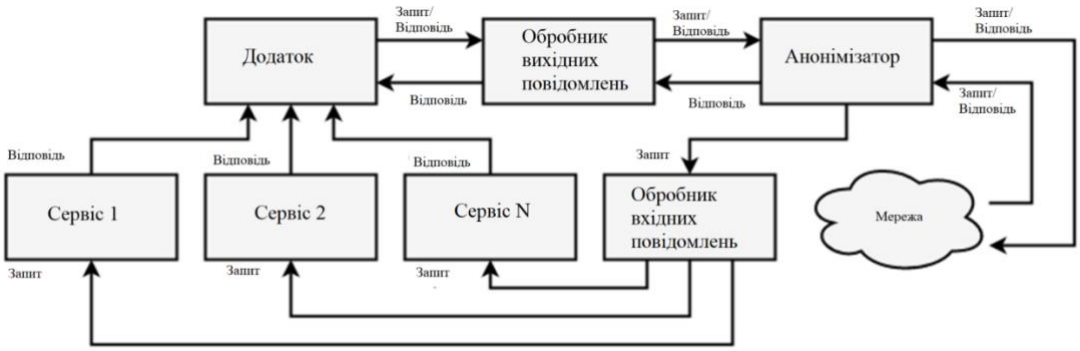


Рисунок 2.32 – Структура роботи сервісу

Анонімізатором, обробником вихідних та вхідних повідомлень є сам HLS. Програми та послуги (як надбудови) вже можна представляти як HLM.

Таким чином, HLS є досить простою і легкою архітектурою ядра анонімної мережі з теоретично доведеною анонімністю. Безперечно мінуси HLS існують і навіть певною мірою більш вирішальні в практичному невикористанні рядовими користувачами. Тим не менш, метою була

реалізація простого, мінімалістичного ядра та розгляд можливостей у подальшому його застосуванні. Із цього народилась реалізація месенджера.

2.6 Реалізація користувацького інтерфейсу

Тепер перейдемо до реалізації самого месенджера. Дозвольте зазначити, що мої компетенції не охоплюють широкого спектру фронтенд розробки, тому було використано існуючі інструменти та ресурси, такі як bootstrap, jquery та їх шаблони [8]. Завдяки застосуванню браузерного середовища для реалізації GUI, вдалося створити інтерфейс, який є доступним та зручним для користувачів. Попри обмежені можливості в моїх знань в цій області, вдалося створити задовільний результат, враховуючи обмежені ресурси та часові рамки проекту.

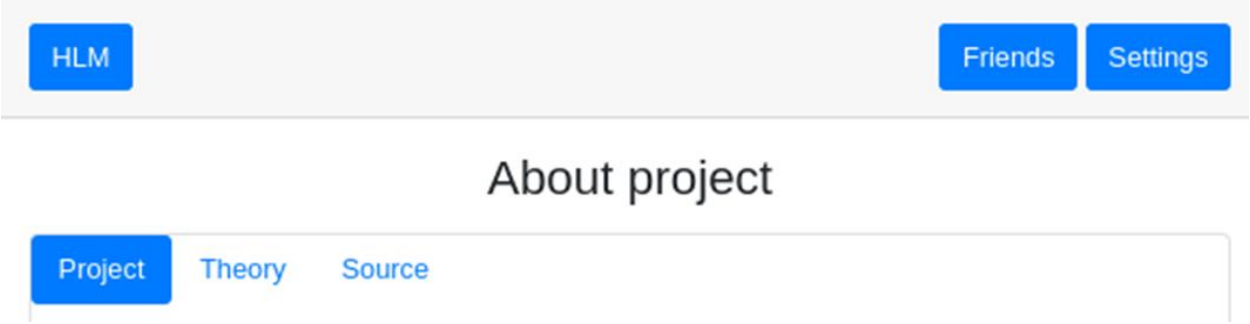


Рисунок 2.33 – Про веб ресурс

Здебільшого засіб анонімного листування, буде лише інтерфейсом, а точніше GUI для реалізованого раніше HLS з прив'язкою до його API.

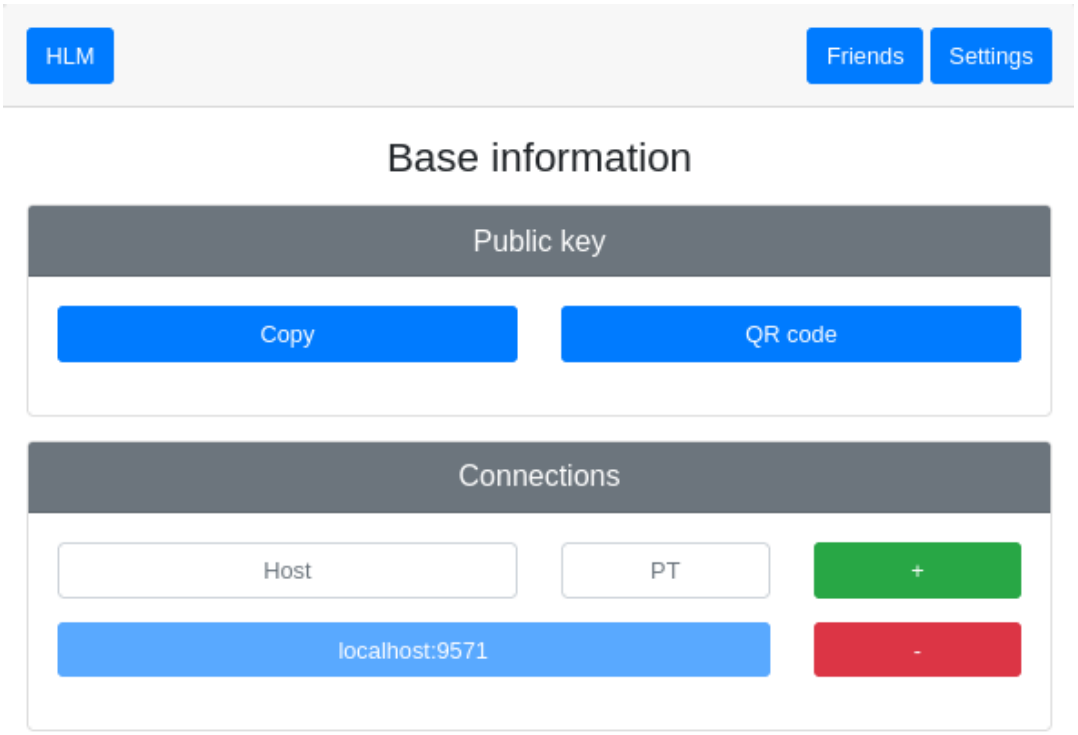


Рисунок 2.34 – Головне вікно веб засобу

Сторінка /settings дозволяє виконати базові налаштування.

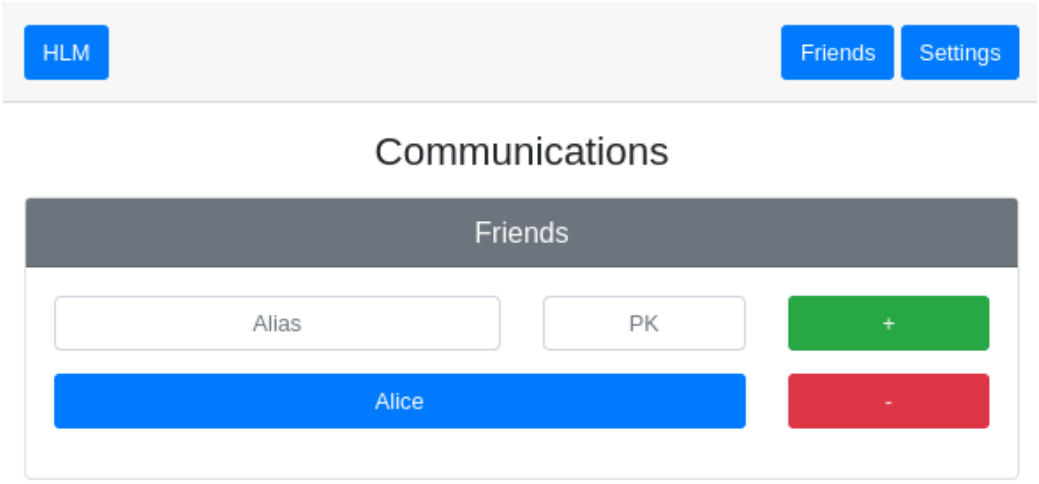


Рисунок 2.35 – Сторінка комунікації

Усі дії месенджера – це всі дії HLS. Отримання, додавання, видалення друзів – це функції API HLS `/api/config/friends`. Отримання, додавання, видалення з'єднань – це також функції API HLS

`/api/config/connects`. Надсилання повідомлень – це функція POST `/api/network/push`. Навіть отримання публічного ключа – це функція `/api/node/pubkey`.

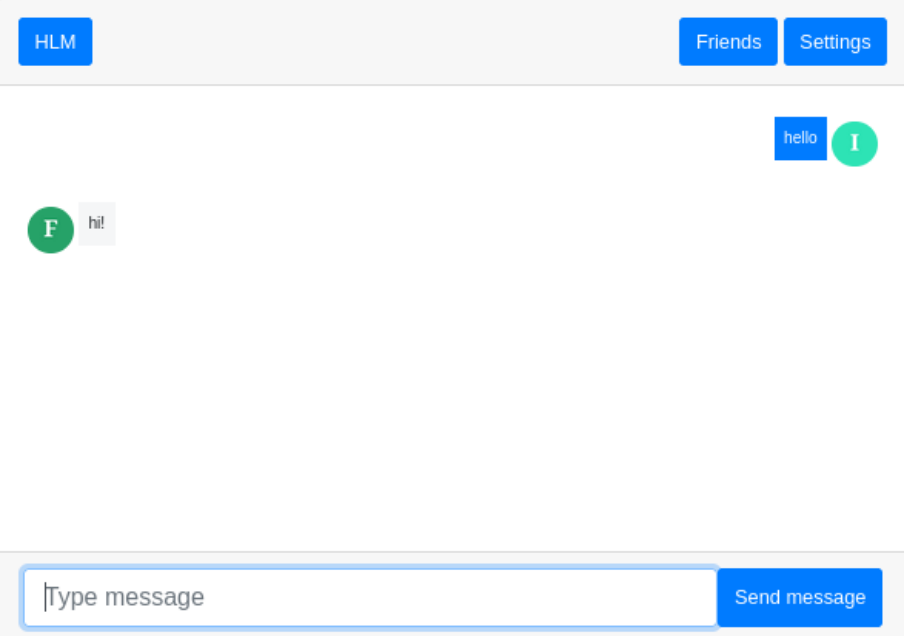


Рисунок 2.36 – Сторінка /friends/chat?alias_name=Alice

Єдина самостійна частина в HLM – це база даних, яка від бази HLS, оскільки першої необхідно зберігати повідомлення, другий лише хеші під час передачі різних повідомлень.

Для отримання повідомлень ззовні HLM створює окремий сервіс для перенаправлення повідомлень із HLS. Код його хендлера показано нижче на рисунку 2.37.

```
890  func HandleIncomigHTTP(db database.IKeyValueDB) http.HandlerFunc {
891  return func(w http.ResponseWriter, r *http.Request) {
892  if r.Method != "POST" {
893  response(w, hls_settings.CErrorMethod, "failed: incorrect method")
894  return
895  }
```

Рисунок 2.37 – Функція HendleIncoming

Щоб подивитися працездатність месенджера, можна перейти до директорії де він знаходиться .

```
$ cd examples/cmd/anon_messenger
$ make
> # Відкриється  два HTTP порта :7070, :8080;
```

У цьому прикладі розгортається три вузли – два учасники чату та один проміжний (створений виключно для маршрутизації).

2.7 Висновок до розділу

У цій частині кваліфікаційної роботи детально розглянуто концепцію створення мережевого ресурсу для анонімного спілкування та успішно реалізовано основні компоненти системи. Також проведено ознайомлення з технологією HTTP Live Streaming (HLS), яка використовується для надання потокових даних і збільшення швидкості завантаження та відтворення контенту.

Одним із ключових кроків у розробці системи стала реалізація мережевої взаємодії кількох клієнтів, що дає змогу забезпечити можливість одночасної передачі анонімних повідомлень у режимі реального часу. Це допомагає створити ефективне середовище для анонімного спілкування між користувачами.

Крім того, в рамках цього розділу розроблено інтерфейс користувача, який відповідає вимогам простоти та зручності використання. Це дозволяє користувачам легко адаптуватися до системи та ефективно використовувати її функції анонімного спілкування. В результаті проведеного дослідження та розробки в рамках розділу "Проектування веб-ресурсу для анонімного листування" було успішно впроваджено ключові компоненти системи та забезпечено взаємодію між користувачами.

3 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

3.1 Актуальність безпеки життєдіяльності

Актуальність безпеки життєдіяльності, зокрема для працівників, які використовують комп'ютери, набуває все більшого значення у сучасному світі. За останні десятиліття з появою та широким поширенням комп'ютерів, поняття безпеки пройшло значну трансформацію. Якщо раніше безпека була пов'язана переважно з фізичною працею або роботою з небезпечним обладнанням, то сьогодні ця проблематика включає в себе і ризики, пов'язані з використанням комп'ютерів [30].

Робота за комп'ютером надає багато переваг, таких як швидкість обробки інформації, зручність спілкування та доступ до великого обсягу даних. Однак, цей спосіб праці також вносить свої виклики та ризики для здоров'я працівників. Однією з найпоширеніших проблем є зорове напруження, яке виникає внаслідок тривалого перебування перед екраном комп'ютера. Часте спостереження за монітором може призвести до втоми очей, сухості та подразнення очей, а також погіршення зору. Це стає особливо актуальним у зв'язку зі зростанням кількості людей, які проводять багато годин на робочому місці перед комп'ютером. Для запобігання цим проблемам рекомендується регулярно робити перерви під час роботи за комп'ютером, використовувати спеціальні окуляри або екранні фільтри, а також забезпечувати належну освітленість приміщення.

Крім того, м'язові напруження є ще одним поширеним проблемою, пов'язаною з роботою за ПК. Довге сидіння перед комп'ютером може призводити до болю в спині, шиї, плечах та зап'ястях. Неправильна позиція тіла, неергономічне обладнання та недостатня фізична активність можуть сприяти розвитку цих проблем. Для зменшення ризику м'язових напружень рекомендується правильно налаштовувати робоче місце, використовувати

ергономічні меблі та аксесуари, регулярно робити паузи для фізичних вправ та розтяжки.

Удосконалення технологій та збільшення їх доступності привели до того, що комп'ютери стали необхідними інструментами у багатьох професіях. Однак, це також приводить до зростання проблем, пов'язаних з безпекою праці. Розлади сну – ще одна проблема, з якою стикаються працівники, які використовують комп'ютери на протязі тривалого часу, особливо пізно ввечері. Екрани комп'ютерів випромінюють світло, яке може впливати на режим сну, призводити до затримки засипання та порушення якості сну. Для покращення якості сну рекомендується обмежувати використання комп'ютера перед сном, встановлювати фільтри на екрани, які зменшують шкідливе випромінювання світла [27].

Крім фізичних проблем, робота за комп'ютером може викликати й психологічний стрес. Постійні терміни, великий обсяг інформації та постійний доступ до роботи можуть призводити до перенапруження та стресу. Висока швидкість реакції, вимоги до продуктивності та постійний потік інформації можуть стати причиною зниження психологічного комфорту та загрози для ментального здоров'я. Для зменшення психологічного стресу рекомендується встановлювати здоровий баланс між роботою та особистим життям, регулярно займатися фізичними вправами, практикувати релаксаційні техніки та брати регулярні перерви від роботи.

Усі ці проблеми і ризики підкреслюють необхідність надати належну увагу безпеці та здоров'ю працівників, які використовують комп'ютери. Наявність належних знань та навичок щодо ергономічного використання комп'ютерів, свідоме виконання рекомендацій щодо безпеки, а також здатність розпізнати та вирішувати проблеми, пов'язані з безпекою роботи за комп'ютером, може значно покращити здоров'я та благополуччя працівників.

Таким чином, безпека життєдіяльності, зокрема для працівників, які використовують комп'ютери, є надзвичайно актуальною сьогодні. Розуміння і усвідомлення проблем, пов'язаних з роботою за комп'ютерами, а також

вживання відповідних заходів для їх уникнення та попередження, відіграють важливу роль у забезпеченні безпеки та здоров'я працівників у сучасному цифровому середовищі.

3.2 Загальні вимоги безпеки з охорони праці для користувачів ПК

Використання персональних комп'ютерів надзвичайно важливо для підвищення продуктивності працівників. Однак, необхідно звернути увагу на те, що така робота може негативно впливати на здоров'я працівників, особливо якщо вона відбувається протягом тривалого періоду або становить значну частину робочого часу.

Взаємодія з персональним комп'ютером може призводити до тимчасового погіршення здоров'я та зниження працездатності працівників. Наприклад, люди, які проводять багато часу за комп'ютером, можуть відчувати головні болі, напругу в очах, погіршення зору, загальну втомленість та роздратованість. Робота з комп'ютером також може спричиняти безсоння, болі у суглобах, спині та руках.

У зв'язку з цим, існують вимоги з охорони праці, які є обов'язковими для всіх роботодавців, незалежно від виду діяльності та форми власності. Ці вимоги передбачають, що працівники, які працюють з комп'ютерами, повинні пройти медичний огляд та інструктаж з охорони праці відповідно до законодавства.

Окремі рекомендації стосуються жінок, які перебувають у період вагітності або годують дитину грудьми. Для них рекомендується обмежити час роботи за комп'ютером до 3 годин на зміну, з урахуванням необхідних перерв та створення оптимальних умов праці відповідно до законодавства [29]. Якщо неможливо організувати роботу з комп'ютером відповідно до цих вимог, жінкам у період вагітності та годування дитини грудьми повинні бути надані роботи, які не пов'язані з використанням комп'ютера.

Законодавство передбачає перелік різних шкідливих та небезпечних виробничих факторів, які можуть впливати на працівників, що працюють з комп'ютерами. Серед них зазначаються підвищений рівень електромагнітних випромінювань, статична електрика, напруженість електричного поля, недостатнє освітлення, монотонність роботи та інші фактори.

Для забезпечення безпечної роботи на комп'ютері працівникам рекомендується пройти медичний огляд, отримати навчання з охорони праці та безпеки роботи, а також мати необхідні знання про шкідливі та небезпечні фактори, які можуть впливати на їхнє здоров'я.

Основні вимоги безпеки з охорони праці для користувачів персональних комп'ютерів повинні включати наступні аспекти:

1. Медичний огляд і інструктаж: Всі працівники, які працюють з комп'ютерами, повинні пройти медичний огляд з метою виявлення можливих проблем зі здоров'ям, пов'язаних з цією роботою. Крім того, їм необхідно отримати інструктаж з охорони праці, який надасть необхідні знання про безпеку роботи з комп'ютером.
2. Обмеження робочого часу: Для жінок, які перебувають у період вагітності або годують дитину грудьми, рекомендується обмежити робочий час з комп'ютером до 3 годин на зміну. Це допоможе запобігти негативному впливу комп'ютерної роботи на їхнє здоров'я. Передбачаються також регулярні перерви та забезпечення оптимальних умов праці.
3. Альтернативні завдання: Якщо неможливо організувати роботу з комп'ютером відповідно до вимог, жінкам у період вагітності та годування дитини грудьми повинні бути надані альтернативні завдання, які не пов'язані з використанням комп'ютера.
4. Виробничі фактори: Робота з комп'ютером може бути пов'язана з різними шкідливими та небезпечними факторами, такими як електромагнітні випромінювання, статична електрика, напруженість електричного поля, недостатнє освітлення та монотонність роботи.

Працівники повинні мати достатні знання про ці фактори та вміти вживати заходи безпеки для їх уникнення.

5. Навчання та свідомість: Регулярне навчання з охорони праці та безпеки роботи на комп'ютері є важливим елементом забезпечення безпеки працівників. Працівники повинні бути обізнані на рахунок можливих ризиків та знати, як їх уникнути або пом'якшити.

3.3 Висновок до розділу

Актуальність безпеки життєдіяльності для працівників, які використовують комп'ютери, зростає у сучасному світі. З появою та широким поширенням комп'ютерів, безпека життєдіяльності пройшла значну трансформацію, охоплюючи ризики, пов'язані з їх використанням. Розуміння цих проблем і вживання відповідних заходів для їх уникнення і попередження відіграють важливу роль у забезпеченні безпеки та здоров'я працівників у цифровому середовищі.

Використання персональних комп'ютерів є важливим для підвищення продуктивності працівників, але вони можуть негативно впливати на здоров'я працівників, особливо при тривалому використанні. Робота за комп'ютером може призводити до тимчасового погіршення здоров'я, втоми та інших негативних ефектів. З метою забезпечення безпеки працівників, існують загальні вимоги безпеки з охорони праці, які передбачають медичний огляд та інструктаж. Законодавство також враховує особливості жінок у період вагітності та годування грудьми, рекомендуючи обмежити час роботи за комп'ютером та надати альтернативні завдання, якщо необхідно.

Окрім того, існує перелік шкідливих та небезпечних виробничих факторів, які можуть впливати на працівників, що працюють з комп'ютерами. Ці фактори включають електромагнітні випромінювання, статичну електрику, недостатнє освітлення, монотонність роботи та інші. Врахування цих факторів

та вживання заходів для зменшення їх впливу є важливим аспектом забезпечення безпеки працівників.

Таким чином, безпека життєдіяльності працівників, які використовують комп'ютери, має високу актуальність у сучасному світі. Врахування ризиків та вживання заходів для забезпечення безпеки та здоров'я працівників є необхідними для успішної роботи в цифровому середовищі.

ВИСНОВКИ

Проведено дослідження предметної області та засобів анонімного листування в реальному часі, що дало можливість виділити їхні переваги та недолік та проаналізувати існуючі протоколи анонімного листування.

Проаналізовано існуючі засоби анонімного спілкування для операційної системи Linux.

Проаналізовано існуючі засоби анонімного спілкування для операційної системи Windows.

Розроблено концепцію засобу анонімного листування та встановлено зв’язки між розробленими модулями для подальшої реалізації основних функцій.

Проаналізовано технологію HLS та її можливості щодо забезпечення анонімності відправки повідомлень та маршрутизації.

Досліджено можливості анонімної маршрутизації та основні принципи її виконання для забезпечення анонімності клієнтів засобу листування.

Проведено розробку основних компонент для реалізації системи анонімного листування.

Проведено реалізацію програмних засобів для мережевої взаємодії.

Розроблено архітектуру засобу анонімного листування та реалізовано веб орієнтований додаток з графічним інтерфейсом реалізованих за допомогою bootstrap та jquery.

ПЕРЕЛІК ДЖЕРЕЛ

1. A New Cell Counter Based Attack Against Tor [Електронний ресурс] / [Z. Ling, J. Luo, W. Yu та ін.]. – 2009. – URL: http://web.cse.ohio-state.edu/~xuan.3/papers/09_ccs_llyfxj.pdf.
2. A Peel of Onion [Електронний ресурс] / Paul Syverson. – 2011. – URL: https://www.acsac.org/2011/program/keynotes/syverson.pdf.
3. A Practical Con-gestion Attack on Tor Using Long Paths [Електронний ресурс] / N. S. Evans, R. Dingledine, C. Grothoff. – 2009. – URL: https://www.freehaven.net/anonbib/cache/congestionlongpaths.pdf.
4. A Stealthy Attack Against Tor Guard Selection [Електронний ресурс] / Q. Li, P. Liu, Z. Qin. – 2015. – URL: https://pdfs.semanticscholar.org/973a/3ad736c743cb50eacccbfb03e275f8ed2e10.pdf.
5. About - Tox [Електронний ресурс] – URL: https://tox.chat/about.html.
6. AL-DHIEF, Fahad Taha, та ін. Performance comparison between TCP and UDP protocols in different simulation scenarios. / International Journal of Engineering & Technology, 2018, 7.4.36: 172-176 с.
7. Browser-Based Attacks on Tor [Електронний ресурс] / T.Abbott, K. Lai, M. Lieberman, E. Price. – 2007. – URL: https://www.freehaven.net/anonbib/cache/abbott-pet2007.pdf.
8. Chernick, M. R. Bootstrap methods: A guide for practitioners and researchers. / John Wiley & Sons, 2011, 400 с..
9. Compromising Anonymity Using Packet Spinning [Електронний ресурс] / [V. Pappas, E. Athanasopoulos, S. Ioannidis та ін.]. – 2008. – URL: https://www.freehaven.net/anonbib/cache/torspinISC08.pdf.
10. DeFabbia-Kane, S. P. Analyzing the effectiveness of passive correlation attacks on the tor anonymity network. / Sam DeFabbia-Kane, 2011, 39 с.

11. End-to-end Encrypted Messaging Protocols: An Overview [Електронний ресурс] / K. Ermoshina, F. Musiani, H. Halpin – 2016. – URL: https://inria.hal.science/hal-01426845/document.
12. How Much Anonymity does Network Latency Leak? [Електронний ресурс] / N. Hopper, E. Y. Vasserman, E. Chan-Tin. – 2010. – URL: https://www-users.cs.umn.edu/~hoppernj/ccs-latency-leak.pdf
13. HTTP Live Streaming [Електронний ресурс] / R. Pantos. – 2017. – URL: https://www.rfc-editor.org/rfc/rfc8216.
14. Low-Cost Traffic Analysis of Tor [Електронний ресурс] / S. J. Murdoch, G. Danezis. – 2005. – URL: https://www.cs.ucy.ac.cy/courses/EPL682/papers/anon-2.pdf.
15. Low-Resource Routing Attacks Against Tor [Електронний ресурс] / [K. Bauer, D. McCoy, D. Grunwald та ін.]. – 2007. – URL: https://www.freehaven.net/anonbib/cache/bauer:wpes2007.pdf.
16. On the Effectiveness of Traffic Analysis Against Anonymity Networks Using Flow Records [Електронний ресурс] / [S. Chakravarty, M. V. Barbera, G. Portokalidis та ін.]. – 2014. – URL: https://www.freehaven.net/anonbib/cache/nfattackpam14.pdf.
17. Onion routing [Електронний ресурс] – URL: http://www.onion-router.net/.
18. Probabilistic Analysis of Onion Routing in a Black-box Model [Електронний ресурс] / J. Feigenbaum, A. Johnson, P. Syverson. – 2012. – URL: https://www.ohmygodel.com/publications/wpes08- feigenbaum.pdf.
19. Recent Attacks On Tor [Електронний ресурс] / Juha Salo. – 2010. – URL: http://www.cse.hut.fi/en/publications/B/11/papers/salo.pdf.
20. Reptile Search Algorithm (RSA): A nature-inspired meta-heuristic optimizer [Електронний ресурс] / L. Abualigah, M. Abd Elaziz, P. Sumari, Z. W. Geem, A. H. Gandomi. – 2022. – URL: https://www.sciencedirect.com/science/article/abs/pii/S0957417421014810.
21. The science of brute force [Електронний ресурс] / M. J. H. Heule, O. Kullmann. – 2017. – URL: https://dl.acm.org/doi/fullHtml/10.1145/3107239.

22. Tor [Електронний ресурс] – URL: http://www.torproject.org/.
23. Tor Metrics [Електронний ресурс] – URL: https://metrics.torproject.org/.
24. Tor Network Status: TorStatus [Електронний ресурс] – URL: https://torstatus.blutmagie.de/.
25. Tor: The Second-Generation Onion Router [Електронний ресурс] / R. Dingledine, N. Mathewson, P. Syverson. – 2004. – URL: https://svn.torproject.org/svn/projects/design-paper/tor-design.pdf.
26. Б. Ю. Жураковський, І. О. Зенів. Комп'ютерні Мережі Частина 1 Навчальний Посібник. / КПІ ім. Ігоря Сікорського, 2020, 328 с.
27. Вплив комп'ютерних технологій на здоров'я людини [Електронний ресурс] / К. С. Холод. – 2020. – URL: http://dspace.pnpu.edu.ua/bitstream/123456789/14978/1/184.pdf
28. Квєтний, Р. Н., Титарчук, Є. О., Гуржій, А. А.. Метод та алгоритм обміну ключами серед груп користувачів на основі асиметричних шифрів ECC та RSA. / Інформаційні технології та комп'ютерна інженерія, 2016, с. 37(3), 38-43.
29. Лісова Є. М., Шарун С. Н., Мартинова С. М.. Вплив хвильового випромінювання на вагітних та розвиток плоду. / Актуальні проблеми експериментальної та клінічної біохімії : матеріали науково-практичної конференції з міжнародною участю, 2018, с. 34.
30. Рогуля, А. О. Еволюція феномена безпеки життєдіяльності. Демократичне врядування. / 2018, 21 с..
31. Українські IT-експерти назвали найбезпечніші месенджери [Електронний ресурс] – URL: https://hmarochos.kiev.ua/2022/04/12/ukrayinski-it-eksperty-nazvaly-najbezpechnishi-mesendzhery/
32. Філіп'єва, М. В.; Гвоздецька, К. П. Порівняння Симетричного І Асиметричного Шифрування. / Міжнародна наукова інтернет-конференція" Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення, 2021, 71 с..

ДОДАТКИ

Додаток А.

Реалізація API засобу анонімного листування

```
package main

import (
	"fmt"

	" /go-peer/modules/client"
	" /go-peer/modules/crypto/asymmetric"
	" /go-peer/modules/payload"
)

func main() {
	var (
		client1 = newClient()
		client2 = newClient()
	)

	msg, err := client1.Encrypt(
		client2.PubKey(),
		payload.NewPayload(0x0, []byte("hello, world!")),
	)
	if err != nil {
		panic(err)
	}

	pubKey, pld, err := client2.Decrypt(msg)
	if err != nil {
		panic(err)
	}

	fmt.Printf("Message: '%s';\nSender's public key: '%s';\n", string(pld.Body()),
pubKey.String())
	fmt.Printf("Encrypted message: '%s'\n", string(msg.ToBytes()))
}

func newClient() client.IClient {
	return client.NewClient(
		client.NewSettings(&client.SSettings{
			FMessageSize: (1 << 12),
		}),
		asymmetric.NewRSAPrivKey(1024),
	)
}
func (node *sNode) Broadcast(pld payload.IPayload) error {
   hash := hashing.NewSHA256Hasher(pld.ToBytes()).Bytes()
	node.inMappingWithSet(hash)

	var err error
```

```
		for _, conn := range node.Connections() {
			e := conn.Write(pld)
			if e != nil {
				err = e
			}
		}

		return err
}
func (conn *sConn) Write(pld payload.IPayload) error {
		conn.fMutex.Lock()
		defer conn.fMutex.Unlock()

		msg := message.NewMessage(pld, []byte(conn.fSettings.GetNetworkKey()))
	packBytes := message.NewPackage(msg.ToBytes()).ToBytes()
		ptr := len(packBytes)
		for {
			n, err := conn.fSocket.Write(packBytes[:ptr])
			if err != nil {
				return err
			}

			ptr = ptr - n
		packBytes = packBytes[:ptr]

			if ptr == 0 {
				break
			}
		}

		return nil
}
func (conn *sConn) Request(pld payload.IPayload) (payload.IPayload, error) {
		var (
			chPld    = make(chan payload.IPayload)
			timeWait = conn.fSettings.GetTimeWait()
		)

		if err := conn.Write(pld); err != nil {
			return nil, err
		}
		go readPayload(conn, chPld)

		select {
		case rpld := <-chPld:
			if rpld == nil {
				return nil, fmt.Errorf("failed: read payload")
			}
			return rpld, nil
		case <-time.After(timeWait):
			return nil, fmt.Errorf("failed: time out")
		}
```

```
}
func readPayload(conn *sConn, chPld chan payload.IPayload) {
   var pld payload.IPayload
	defer func() {
		chPld <- pld
	}()

	bufLen := make([]byte, encoding.CSizeUint64)
	length, err := conn.fSocket.Read(bufLen)
	if err != nil {
		return
	}
	if length != encoding.CSizeUint64 {
		return
	}

	arrLen := [encoding.CSizeUint64]byte{}
	copy(arrLen[:], bufLen)

	mustLen := encoding.BytesToUint64(arrLen)
	if mustLen > conn.fSettings.GetMessageSize() {
		return
	}

	msgRaw := make([]byte, 0, mustLen)
	for {
		buffer := make([]byte, mustLen)
		n, err := conn.fSocket.Read(buffer)
		if err != nil {
			return
		}

		msgRaw = bytes.Join(
			[][]byte{
				msgRaw,
				buffer[:n],
			},
			[]byte{},
		)

		mustLen -= uint64(n)
		if mustLen == 0 {
			break
		}
	}

	msg := message.LoadMessage(
		msgRaw,
		[]byte(conn.fSettings.GetNetworkKey()),
	)
	if msg == nil {
		return
```

```
		}

		pld = msg.Payload()
	}
	type IHandlerF func(INode, conn.IConn, payload.IPayload)

	func (node *sNode) Handle(head uint64, handle IHandlerF) INode {
		node.fMutex.Lock()
		defer node.fMutex.Unlock()

		node.fHandleRoutes[head] = handle
		return node
	}
	func (node *sNode) handleConn(address string, conn conn.IConn) {
		defer node.Disconnect(address)
		for {
			ok := node.handleMessage(conn, conn.Read())
			if !ok {
				break
			}
		}
	}
	func (node *sNode) handleMessage(conn conn.IConn, pld payload.IPayload) bool {
		if pld == nil {
			return false
		}

		hash := hashing.NewSHA256Hasher(pld.ToBytes()).Bytes()
		if node.inMappingWithSet(hash) {
			return true
		}

		f, ok := node.getFunction(pld.Head())
		if !ok || f == nil {
			return false
		}

		f(node, conn, pld)
		return true
	}
	func (conn *sConn) Read() payload.IPayload {
		chPld := make(chan payload.IPayload)
		go readPayload(conn, chPld)
		return <-chPld
	}
	package main

	import (
		"fmt"
		"strconv"
		"time"
```

```
	" /go-peer/modules/network"
	" /go-peer/modules/network/conn"
	" /go-peer/modules/payload"
)

const (
	serviceHeader  = 0xDEADBEAF
	serviceAddress = ":8080"
)

func main() {
	var (
		service1 = network.NewNode(network.NewSettings(&network.SSettings{}))
		service2 = network.NewNode(network.NewSettings(&network.SSettings{}))
	)

	service1.Handle(serviceHeader, handler("#1"))
	service2.Handle(serviceHeader, handler("#2"))

	go service1.Listen(serviceAddress)
	time.Sleep(time.Second) // wait

	_, err := service2.Connect(serviceAddress)
	if err != nil {
		panic(err)
	}

	service2.Broadcast(payload.NewPayload(
		serviceHeader,
		[]byte("0"),
	))

	select {}
}

func handler(serviceName string) network.IHandlerF {
	return func(n network.INode, c conn.IConn, p payload.IPayload) {
		time.Sleep(time.Second) // delay for view "ping-pong" game

		num, err := strconv.Atoi(string(p.Body()))
		if err != nil {
			panic(err)
		}

		val := "ping"
		if num%2 == 1 {
			val = "pong"
		}

		fmt.Printf("service '%s' got '%s#%d'\n", serviceName, val, num)

		n.Broadcast(payload.NewPayload(
```

```
                        serviceHeader,
                        []byte(fmt.Sprintf("%d", num+1)),
                    ))
            }
    }
    func (q *sQueue) Enqueue(msg message.IMessage) error {
            q.fMutex.Lock()
            defer q.fMutex.Unlock()

            if uint64(len(q.fQueue)) >= q.Settings().GetCapacity() {
                    return errors.New("queue already full, need wait and retry")
            }

            go func() {
                    q.fMutex.Lock()
                    defer q.fMutex.Unlock()

                    q.fQueue <- msg
            }()

            return nil
    }
    func (q *sQueue) Dequeue() <-chan message.IMessage {
            time.Sleep(q.Settings().GetDuration())

            go func() {
                    q.fMutex.Lock()
                    defer q.fMutex.Unlock()

                    if !q.fIsRun {
                            return
                    }

                    if len(q.fQueue) == 0 {
                            q.fQueue <- (<-q.fMsgPull.fQueue)
                    }
            }()

            return q.fQueue
    }
    type IHandlerF func(INode, asymmetric.IPubKey, payload.IPayload) []byte

    func (node *sNode) Handle(head uint32, handle IHandlerF) INode {
            node.fMutex.Lock()
            defer node.fMutex.Unlock()

            node.fHandleRoutes[head] = handle
            return node
    }
    func (node *sNode) handleWrapper() network.IHandlerF {
            go func() {
                    for {
```

```
                msg, ok := <-node.Queue().Dequeue()
                if !ok {
                        break
                }
                node.broadcast(msg)
        }
    }()

    return func(nnode network.INode, _ conn.IConn, npld payload.IPayload) {
msg := node.initialCheck(message.LoadMessage(npld.Body()))
        if msg == nil {
                return
        }

        nnode.Broadcast(npld)
        client := node.Queue().Client()

        sender, pld, err := client.Decrypt(msg)
        if err != nil {
                return
        }

        if !node.F2F().InList(sender) {
                return
        }

hash := []byte(fmt.Sprintf("_hash_%X", msg.Body().Hash()))
        if _, err := node.KeyValueDB().Get(hash); err == nil {
                return
        }
        node.KeyValueDB().Set(hash, []byte{})

        head := pld.Head()
        action, ok := node.getAction(
                loadHead(head).Actions(),
        )
        if ok {
                action <- pld.Body()
                return
        }

        f, ok := node.getRoute(
                loadHead(head).Routes(),
        )
        if !ok || f == nil {
                return
        }

        resp := f(node, sender, pld)
if resp == nil {
                return
        }
```

```
                respMsg, err := client.Encrypt(
                        sender,
                        payload.NewPayload(head, resp),
                )
                if err != nil {
                        panic(err)
                }

                for i := uint64(0); i <= node.Settings().GetRetryEnqueue(); i++ {
                        err := node.Queue().Enqueue(respMsg)
                        if err != nil {
                                time.Sleep(node.Queue().Settings().GetDuration())
                                continue
                        }
                        break
                }
        }
}
func (node *sNode) broadcast(msg message.IMessage) error {
        return node.Network().Broadcast(payload.NewPayload(
                settings.CMaskNetwork,
                msg.ToBytes(),
        ))
}
func (node *sNode) initialCheck(msg message.IMessage) message.IMessage {
   if msg == nil {
                return nil
        }
        if len(msg.Body().Hash()) != hashing.CSHA256Size {
                return nil
        }
        diff := node.Queue().Client().Settings().GetWorkSize()
        puzzle := puzzle.NewPoWPuzzle(diff)
        if !puzzle.Verify(msg.Body().Hash(), msg.Body().Proof()) {
                return nil
        }

        return msg
}
func (node *sNode) Request(recv asymmetric.IPubKey, pld payload_adapter.IPayload) ([]byte,
error) {
   if len(node.Network().Connections()) == 0 {
                return nil, errors.New("length of connections = 0")
        }
        headAction := uint32(random.NewStdPRNG().Uint64())
        headRoutes := mustBeUint32(pld.Head())

        newPld = payload.NewPayload(
                joinHead(headAction, headRoutes).Uint64(),
                pld.Body(),
        )
```

```
	msg, err := node.Queue().Client().Encrypt(recv, newPld)
	if err != nil {
		return nil, err
	}
	node.setAction(headAction)
	defer node.delAction(headAction)
	for i := uint64(0); i <= node.Settings().GetRetryEnqueue(); i++ {
		if err := node.Queue().Enqueue(msg); err != nil {
			time.Sleep(node.Queue().Settings().GetDuration())
			continue
		}
		break
	}
	return node.recv(headAction, node.Settings().GetTimeWait())
}
func (node *sNode) recv(head uint32, timeOut time.Duration) ([]byte, error) {
   action, ok := node.getAction(head)
	if !ok {
		return nil, errors.New("action undefined")
	}
	select {
	case result, opened := <-action:
		if !opened {
			return nil, errors.New("chan is closed")
		}
		return result, nil
	case <-time.After(timeOut):
		return nil, errors.New("time is over")
	}
}
package main

import (
	"fmt"
	"os"
	"strconv"
	"time"

	" /go-peer/modules/client"
	" /go-peer/modules/crypto/asymmetric"
	" /go-peer/modules/friends"
	" /go-peer/modules/network"
	" /go-peer/modules/network/anonymity"
	payload_adapter " /go-peer/modules/network/anonymity/adapters/payload"
	" /go-peer/modules/network/conn"
	" /go-peer/modules/payload"
	" /go-peer/modules/queue"
	" /go-peer/modules/storage/database"
)

const (
	serviceHeader  = 0xDEADBEAF
```

```
	serviceAddress = ":8080"
)

const (
	dbPath1 = "database1.db"
	dbPath2 = "database2.db"
)

func deleteDBs() {
	os.RemoveAll(dbPath1)
	os.RemoveAll(dbPath2)
}

func main() {
	deleteDBs()
	defer deleteDBs()

	var (
		service1 = newNode(dbPath1)
		service2 = newNode(dbPath2)
	)

	service1.Handle(serviceHeader, handler("#1"))
	service2.Handle(serviceHeader, handler("#2"))
	service1.F2F().Append(service2.Queue().Client().PubKey())
	service2.F2F().Append(service1.Queue().Client().PubKey())
	if err := service1.Run(); err != nil {
		panic(err)
	}
	if err := service2.Run(); err != nil {
		panic(err)
	}

	go service1.Network().Listen(serviceAddress)
	time.Sleep(time.Second)

	if _, err := service2.Network().Connect(serviceAddress); err != nil {
		panic(err)
	}
	msg, err := service2.Queue().Client().Encrypt(
		service1.Queue().Client().PubKey(),
		payload_adapter.NewPayload(
			serviceHeader,
			[]byte("0"),
		),
	)
	if err != nil {
		panic(err)
	}
	if err := service2.Queue().Enqueue(msg); err != nil {
		panic(err)
	}
```

```
	select {}
}

func handler(serviceName string) anonymity.IHandlerF {
	return func(node anonymity.INode, pubKey asymmetric.IPubKey, pld
payload.IPayload) []byte {
		num, err := strconv.Atoi(string(pld.Body()))
		if err != nil {
			panic(err)
		}

		val := "ping"
		if num%2 == 1 {
			val = "pong"
		}

		fmt.Printf("service '%s' got '%s#%d'\n", serviceName, val, num)

		msg, err := node.Queue().Client().Encrypt(
			pubKey,
			payload_adapter.NewPayload(
				serviceHeader,
				[]byte(fmt.Sprintf("%d", num+1)),
			),
		)
		if err != nil {
			panic(err)
		}

		if err := node.Queue().Enqueue(msg); err != nil {
			panic(err)
		}
		return nil
	}
}

func newNode(dbPath string) anonymity.INode {
	return anonymity.NewNode(
		anonymity.NewSettings(&anonymity.SSettings{}),
		database.NewLevelDB(
			database.NewSettings(&database.SSettings{
				FPath: dbPath,
			}),
		),
		network.NewNode(
			network.NewSettings(&network.SSettings{
				FConnSettings: conn.NewSettings(&conn.SSettings{}),
			}),
		),
		queue.NewQueue(
			queue.NewSettings(&queue.SSettings{}),
```

```
                        client.NewClient(
                                client.NewSettings(&client.SSettings{}),
                                asymmetric.NewRSAPrivKey(1024),
                        ),
                ),
                friends.NewF2F(),
        )
}
package main

import (
        "encoding/json"
        "io"
        "net/http"
)

type sResponse struct {
        FEcho   string `json:"echo"`
        FReturn int    `json:"return"`
}

func main() {
        http.HandleFunc("/echo", echoPage)
        http.ListenAndServe(":8080", nil)
}

func echoPage(w http.ResponseWriter, r *http.Request) {
        if r.Method != "POST" {
                response(w, 2, "failed: incorrect method")
                return
        }
        res, err := io.ReadAll(r.Body)
        if err != nil {
                response(w, 3, "failed: read body")
                return
        }
        response(w, 1, string(res))
}

func response(w http.ResponseWriter, ret int, res string) {
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(&sResponse{
                FEcho:   res,
                FReturn: ret,
        })
}
package handler

import (
        "bytes"
        "fmt"
        "io"
```

```
	"net/http"

	" /go-peer/cmd/hls/config"
	hls_network " /go-peer/cmd/hls/network"
	hls_settings " /go-peer/cmd/hls/settings"
	" /go-peer/modules/crypto/asymmetric"
	" /go-peer/modules/network/anonymity"
	" /go-peer/modules/payload"
)
func HandleServiceTCP(cfg config.IConfig) anonymity.IHandlerF {
	return func(node anonymity.INode, sender asymmetric.IPubKey, pld payload.IPayload)
[]byte {
		requestBytes := pld.Body()
		request := hls_network.LoadRequest(requestBytes)
		if request == nil {
			return nil
		}
		address, ok := cfg.Service(request.Host())
		if !ok {
			return nil
		}
		req, err := http.NewRequest(
			request.Method(),
			fmt.Sprintf("http://%s%s", address, request.Path()),
			bytes.NewReader(request.Body()),
		)
		if err != nil {
			return nil
		}
		req.Header.Add(hls_settings.CHeaderPubKey, sender.String())
		for key, val := range request.Head() {
			if key == hls_settings.CHeaderPubKey {
				continue
			}
			req.Header.Add(key, val)
		}
		resp, err := http.DefaultClient.Do(req)
		if err != nil {
			return nil
		}
		defer resp.Body.Close()
		data, err := io.ReadAll(resp.Body)
		if err != nil {
			return nil
		}
		return data
	}
}
```