КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра радіотехніки та радіоелектронних систем

«На правах рукопису»

Робота допущена до захисту в ЕК
рішенням кафедри радіотехніки та радіоелектронних систем
від _____2026 року, протокол № _____
В.о.завідувача кафедри к.фіз.-мат.наук, доцент
_____ Ігор БЕХ

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА
на тему:

«КЛІЄНТ-СЕРВЕРНА СИСТЕМА ДИСТАНЦІЙНОЇ ЕКСПЕРТНОЇ ДОПОМОГИ З ПРОСТОРОВИМИ АНОТАЦІЯМИ У СЕРЕДОВИЩІ ЗМІШАНОЇ РЕАЛЬНОСТІ»

Виконав:
студент 2-го курсу магістратури
денної форми здобуття освіти
спеціальності 172 Електронні комунікації та радіотехніка
Інформаційна безпека телекомунікаційних систем і мереж
Завадський Олександр Олегович_____

Науковий керівник:
к. т. н., с.н.с
Жиров Генадій Борисович _____

Рецензент:
завідувач кафедри комп'ютерних наук
Державного університету інформаційно-комунікаційних технологій,
доктор технічних наук, професор
Вишнівський Віктор Вікторович_____

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент _____ Олександр ЗАВАДСЬКИЙ

Київ - 2026

РЕФЕРАТ

Дипломний проект, 109 с., 01 дод., 48 джерел.

Ключові слова: змішана реальність, дистанційна експертна допомога, просторові анотації, World-Space прив'язка, WEBRTC, UDP-відеопотік, META QUEST 3S, Клієнт-Серверна архітектура, ANDROID RELAY, COTURN TURN-сервер, VI-SLAM, безпека телекомунікаційних систем

Об'єкт дослідження: Клієнт-серверна система дистанційної експертної допомоги з просторовими анотаціями у середовищі змішаної реальності на базі гарнітури Meta Quest 3S.

Мета роботи: Підвищення ефективності просторової взаємодії у системах дистанційної експертної підтримки шляхом розробки та дослідження дворівневої клієнт-серверної архітектури з використанням world-space анотацій у середовищі змішаної реальності.

Основний зміст:

У першому розділі виконано систематичний огляд існуючих AR-систем дистанційної допомоги - Microsoft Dynamics 365 Remote Assist, TeamViewer Assist AR, PTC Vuforia Chalk, Scope AR WorkLink - та виявлено їх спільний недолік: використання screen-space анотацій або залежність від SaaS-інфраструктури; досліджено passthrough-технології MR-гарнітур (OST та VST/Passthrough), протоколи передачі даних UDP, WebSocket і WebRTC, методи просторової реєстрації анотацій (VI-SLAM, depth raycast, OVRSpatialAnchor), стекові рішення для відеостримінгу (HLS, RTMP, SRT, WebRTC, кодеки H.264/JPEG), а також загрози інформаційній безпеці систем реального часу (MITM, DoS, IP leakage, session hijacking) та засоби їх нейтралізації (TLS 1.3, DTLS, SRTP, WSS).

У другому розділі описано архітектуру та реалізацію авторської системи з проектною назвою «V.I.R.A.», що складається з чотирьох компонентів: Unity-застосунку для Meta Quest 3S (відеостримінг JPEG-over-UDP та обробка анотацій), Android Relay (Kotlin, UDP-WebSocket-міст), хмарного Node.js/TypeScript-сервера з coturn TURN-сервером та браузерного інтерфейсу оператора (React/TypeScript з WebRTC DataChannel); детально розглянуто реалізацію кожного компонента,

включно з фрагментацією UDP-пакетів, WebSocket-relay, WebRTC SDP/ICE-handshake та React-інтерфейсом з маркерною палітрою і режимом Freeze Frame.

Розроблено оригінальний механізм world-space просторових анотацій на основі OVRSpatialAnchor (Meta XR SDK) та depth raycast через Meta Environment Depth API: для кожної 2D-координати оператора виконується зворотне проектування у тривимірний простір сцени з точністю 3-5 см при доступному depth-сенсорі або з fallback-глибиною 3,0 м, після чого анотація прив'язується до VI-SLAM-карти гарнітури та зберігає стабільне положення у просторі.

За результатами тестування підтверджено наскрізну затримку 30-80 мс у LAN-режимі (UDP + WebSocket) та 80-200 мс (120-350 мс через TURN relay) у Cloud-режимі (WebRTC через coturn, Hetzner CAX11); завантаження CPU сервера при ретрансляції 18-25 кадрів/с не перевищує 1,5% event loop Node.js, що підтверджує масштабованість IO-bound архітектури; система досягла рівня внутрішнього версіонування v0.7.0 Alpha (40 зібраних білдів (скомпільованих версій програм) з хронологією від 21.11.2025 до 25.03.2026 включно) зі стабільно працюючим LAN-режимом та MVP-готовністю Cloud-режиму.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

V.I.R.A. – Virtual Interactive Remote Assistance (Віртуальна інтерактивна дистанційна допомога)

API - Application Programming Interface (Інтерфейс програмування застосунків)

AR - Augmented Reality (Доповнена реальність)

CPU - Central Processing Unit (Центральний процесор)

DoS - Denial of Service (Відмова в обслуговуванні)

DTLS - Datagram Transport Layer Security (Протокол захисту датаграм транспортного рівня)

FPS - Frames Per Second (Кількість кадрів за секунду)

GPU - Graphics Processing Unit (Графічний процесор)

HMD - Head-Mounted Display (Нашоломний дисплей)

ICE - Interactive Connectivity Establishment (Протокол встановлення інтерактивного з'єднання)

IoT - Internet of Things (Інтернет речей)

JSON - JavaScript Object Notation (Формат обміну даними)

LAN - Local Area Network (Локальна обчислювальна мережа)

MITM - Man-in-the-Middle (Атака «людина посередині»)

MR - Mixed Reality (Змішана реальність)

NAT - Network Address Translation (Трансляція мережевих адрес)

OST - Optical See-Through (Оптична прозорість)

P2P - Peer-to-Peer (Однорангова мережа)

SDK - Software Development Kit (Набір засобів розробника)

SDP - Session Description Protocol (Протокол опису сесії)

SRTP - Secure Real-time Transport Protocol (Захищений протокол передачі реального часу)

STUN - Session Traversal Utilities for NAT (Утиліти проходження сесій через NAT)

TCP - Transmission Control Protocol (Протокол управління передачею)

TURN - Traversal Using Relays around NAT (Проходження NAT з використанням ретрансляторів)

UDP - User Datagram Protocol (Протокол користувацьких датаграм)

VI-SLAM - Visual-Inertial Simultaneous Localization and Mapping (Візуально-інерціальна одночасна локалізація та побудова карти)

VST - Video See-Through (Відеопрозорість)

WebRTC - Web Real-Time Communication (Веб-комунікації реального часу)

WSS - WebSocket Secure (Захищений протокол WebSocket)

ВСТУП

Сучасна промисловість, медицина, інфраструктурна енергетика та інші галузі з підвищеними вимогами до кваліфікації персоналу стикаються зі спільною проблемою: висококваліфікований фахівець не завжди може бути фізично присутнім у місці виникнення нештатної ситуації. Виїзд експерта пов'язаний із витратами часу та коштів, а в умовах роботи у важкодоступних, радіаційно або хімічно небезпечних середовищах - ще й із прямими ризиками для здоров'я. Технології дистанційної підтримки покликані усунути цю суперечність, надаючи змогу фахівцю керувати діями виконавця на відстані в режимі реального часу.

Упродовж останнього десятиліття на ринку з'явилася низка комерційних AR-рішень дистанційної допомоги: Microsoft Dynamics 365 Remote Assist, TeamViewer Assist AR, PTC Vuforia Chalk, Scope AR WorkLink. [4-6] Однак систематичний аналіз цих платформ у межах цієї роботи виявив принципові обмеження, характерні для більшості з них. По-перше, анотації в переважній більшості систем є screen-space: вони накладаються на двовимірне зображення і не прив'язуються до тривимірної геометрії сцени, через що зміщуються або зникають при найменшому русі головою. По-друге, провідні рішення функціонують виключно як SaaS-сервіси у хмарі постачальника (Microsoft Azure, хмара TeamViewer), що унеможливлює їх використання у закритих корпоративних мережах або середовищах без доступу до зовнішнього інтернету. По-третє, ці платформи орієнтовані на гарнітури optical see-through (Microsoft HoloLens 2, Magic Leap) або мобільні пристрої з ARCore/ARKit, тоді як гарнітури з video passthrough, зокрема Meta Quest 3S, є доступнішими за вартістю та універсальнішими у застосуванні, але зазвичай не підтримуються.

Паралельно з розвитком комерційних продуктів у науковій літературі активно досліджують методи підвищення якості дистанційної AR-взаємодії [14]. Зокрема, в оглядовій статті «A Survey on Synchronous Augmented, Virtual, and Mixed Reality Remote Collaboration Systems» [1], що охоплює 87 публікацій за 25 років, систематизовано підходи до класифікації анотацій (screen-space vs. world-space),

методів відстеження (SLAM, маркери, depth-сенсори) і транспортних протоколів. Дослідження CPR Emergency Assistance Through Mixed Reality Communication та проєкт ARRA (Augmented Reality for Remote Assistance, Cranfield University, 60 учасників) [2] підтверджують, що world-space анотації статистично значущо скорочують час виконання завдання порівняно зі screen-space підходом. Водночас більшість академічних прототипів мають суттєві обмеження з точки зору практичного впровадження: залежність від зовнішньої хмарної інфраструктури, відсутність підтримки двох транспортних режимів (LAN і Cloud), а також брак інтеграції з гарнітурами класу standalone passthrough MR.

Актуальність теми визначається кількома чинниками. З технологічного боку, Meta Quest 3S [12] є першою масово доступною standalone MR-гарнітурою з кольоровим RGB passthrough 18 PPD, Qualcomm Snapdragon XR2 Gen 2 та повноцінним Meta Environment Depth API, що відкриває якісно нові можливості для побудови практичних world-space AR-систем без дорогих стаціонарних рішень. З інфраструктурного боку, зростання вимог до кібербезпеки та суверенітету даних зумовлює попит на self-hosted рішення, які можна розгорнути в ізольованому контурі підприємства. З наукового погляду, поєднання UDP-відеостримінгу, WebRTC з TURN-relay та анотацій, прив'язаних через OVRSpatialAnchor, в єдиній системі з двома транспортними режимами не має готових відкритих реалізацій і потребує комплексного дослідження.

Основна науково-технічна задача роботи полягає у створенні методів і алгоритмів надійної передачі відеопотоку реального часу та просторового суміщення двовимірних керуючих команд оператора із тривимірним середовищем змішаної реальності в умовах гетерогенних мереж. Розв'язання цієї задачі, а також розробка відповідної self-hosted архітектури для гарнітури Meta Quest 3S, відповідає сучасному стану розвитку телекомунікаційних технологій і потребам практичного застосування в закритих промислових мережах.

Метою роботи є підвищення ефективності просторової взаємодії у системах дистанційної експертної підтримки шляхом розробки та дослідження дворівневої

клієнт-серверної архітектури з використанням world-space анотацій у середовищі змішаної реальності.

Для досягнення поставленої мети сформульовано такі задачі дослідження:

1. Провести систематичний аналіз наявних комерційних і академічних AR/MR-систем дистанційної допомоги та виявити їхні переваги й обмеження з погляду просторових анотацій, транспортних протоколів і моделі розгортання.

2. Дослідити технологічну базу системи: passthrough-характеристики Meta Quest 3S, методи просторової реєстрації (VI-SLAM, depth raycast, OVRSpatialAnchor), протоколи UDP, WebSocket і WebRTC, а також загрози інформаційній безпеці систем реального часу.

3. Розробити архітектуру та реалізувати чотири взаємопов'язані компоненти системи: Unity-застосунок для Meta Quest 3S, Android Relay (Kotlin), хмарний Node.js/TypeScript-сервер із coturn і браузерний React/TypeScript-інтерфейс оператора.

4. Розробити та реалізувати механізм world-space прив'язки анотацій на основі Meta Environment Depth API (depth raycast) та OVRSpatialAnchor із fallback-режимом для умов відсутності даних глибини.

5. Реалізувати два транспортні режими (LAN і Cloud) з єдиним API анотацій і провести порівняльне тестування їхніх характеристик: наскрізної затримки, навантаження на сервер і стійкості з'єднання.

6. Оцінити захищеність системи відповідно до моделі загроз (конфіденційність, цілісність, доступність) і верифікувати застосовані заходи захисту (TLS 1.3, DTLS/SRTP, WSS).

Об'єктом дослідження є клієнт-серверна система дистанційної експертної допомоги з просторовими анотаціями в середовищі змішаної реальності — авторська система з проектною назвою «V.I.R.A.», яка складається з чотирьох програмних компонентів і функціонує у двох транспортних режимах.

Предметом дослідження є методи та принципи побудови такої системи: алгоритми world-space прив'язки 2D-анотацій оператора до тривимірної геометрії сцени засобами depth raycast і VI-SLAM; протоколи реального часу (UDP,

WebSocket, WebRTC) та їхні порівняльні характеристики в контексті низькозатримного відеостримінгу; архітектурні рішення relay-сервера для підтримки LAN- і Cloud-режимів з єдиним програмним інтерфейсом; а також механізми забезпечення інформаційної безпеки (DTLS, SRTP, TLS 1.3, WSS) у системах із гетерогенним транспортом.

У роботі застосовано комплекс взаємодоповнювальних методів дослідження. Системний аналіз і порівняльний метод використано у другому розділі для класифікації наявних AR-систем дистанційної допомоги за типом анотацій, транспортним протоколом і моделлю розгортання, а також для вибору оптимальних технологічних рішень на основі кількісних та якісних показників. Методи об'єктно-орієнтованого та компонентного проєктування програмного забезпечення застосовано під час розроблення архітектури системи: для визначення меж між компонентами, специфікації протоколів міжкомпонентної взаємодії та схем даних (JSON-формат анотацій, UDP-фрагментація, WebRTC SDP/ICE). Методи прикладної математики, зокрема зворотне проектування ray через intrinsic-параметри камери та перетин ray з depth-поверхнею, а також методи аналітичної геометрії використано під час розроблення алгоритму перетворення 2D-координат оператора у world-space-координати сцени. Метод натурного експерименту застосовано для вимірювання ключових метрик системи: наскрізної затримки відеопотоку, навантаження на CPU сервера та точності прив'язки анотацій в умовах, максимально наближених до реальних.

Результатом роботи є повністю функціональна авторська система, придатна до практичного застосування: LAN-режим є production-ready (готовий для використання), а Cloud-режим досяг рівня MVP («Minimal Working Product») з підтвердженою наскрізною затримкою 30--80 мс у LAN та 80--200 мс у Cloud через TURN-relay на Hetzner CAX11 у Франкфурті. Система є self-hosted і не потребує сторонніх SaaS-підписок, що дає змогу розгортати її у закритих корпоративних або відомчих мережах, зокрема на об'єктах критичної інфраструктури, де передавання відеоданих через зовнішні хмарні сервіси є неприйнятним. Розроблені архітектурні рішення, зокрема дворежимний транспортний стек з єдиним JSON API анотацій,

алгоритм world-space-прив'язки через depth raycast з OVRSpatialAnchor та схема Android Relay як UDP - WebSocket-моста, можуть бути використані як самостійні інженерні рішення під час розроблення інших систем MR-взаємодії. Практичну цінність роботи підтверджують і результати тестування: точність просторової прив'язки анотацій становить 3--5 см за активного Meta Environment Depth API, що є достатнім для задач технічного обслуговування, медичної асистенції та навчання персоналу у виробничих умовах.

І. АНАЛІЗ ТЕХНОЛОГІЙ ТА ПРЕДМЕТНОЇ ГАЛУЗІ СИСТЕМ AR-ПІДТРИМКИ

1.1. Системи дистанційної AR-підтримки: класифікація та огляд існуючих рішень

Дослідження доцільно розпочати з визначення предметної галузі. Системи дистанційної підтримки на основі доповненої реальності (AR Remote Assistance Systems) являють собою клас програмно-апаратних комплексів, що забезпечують передачу візуальної та просторової інформації між оператором-експертом та виконавцем на місці події у режимі реального часу. На відміну від традиційних відеоконференційних рішень, такі системи доповнюють відеопотік семантичними або просторовими анотаціями, які накладаються на фізичне оточення виконавця. Це дозволяє усунути неоднозначність усних вказівок і значно підвищити точність передачі просторової інформації. Дослідження, проведені в рамках системи ARRA (Augmented Reality for Remote Assistance), підтвердили, що застосування AR-анотацій у дистанційній підтримці забезпечує покращення точності просторового референсування на 30% порівняно зі стандартними відеодзвінками [2].

Функціональна роль таких систем охоплює широкий спектр застосувань: технічне обслуговування та ремонт обладнання, польовий сервіс, медичні телеконсультації, промислове навчання персоналу, а також підтримку в екстремальних або небезпечних умовах. Ринок програмного забезпечення для AR-підтримки сегментується за компонентами (програмне забезпечення та сервіси), режимами розгортання (хмарний, on-premises), а також за галузями застосування, серед яких домінують виробництво, польовий сервіс і охорона здоров'я.

Важливим аспектом аналізу є таксономія систем. Структурований підхід до класифікації систем дистанційної AR-підтримки запропонований у ґрунтовному огляді «A Survey on Synchronous Augmented, Virtual, and Mixed Reality Remote

Collaboration Systems» [1], де аналізуються 87 унікальних систем, у тому числі 25 комерційних рішень. Автори формують таксономію на основі трьох ключових стовпів: *середовище* (Environment), *аватари* (Avatars) та *взаємодія* (Interaction). У рамках цієї класифікації системи розрізняються за способом представлення спільного робочого простору, ступенем присутності дистанційного учасника та механізмами взаємодії з об'єктами реального світу.

Для цілей цього дослідження доцільно виділити кілька ортогональних ознак класифікації, що відображають архітектурні та функціональні відмінності між існуючими рішеннями:

1. За типом клієнтського пристрою виконавця:

- Смартфон або планшет (Android / iOS) - найдоступніший варіант, проте обмежений у частині hands-free взаємодії
- Окуляри доповненої реальності (optical see-through, OST) - наприклад, Microsoft HoloLens 2, Magic Leap
- Гарнітури змішаної реальності з відеопрозорістю (video passthrough) - наприклад, Meta Quest 3/3S
- Стаціонарні або мобільні камерні системи (360°-telepresence)

2. За типом анотацій і просторового прив'язування:

- Плоскі (screen-space) анотації - накладаються на двовимірне зображення без урахування тривимірної геометрії сцени
- Просторово прив'язані (world-space) анотації - закріплені у тривимірному просторі за допомогою трекінгу або SLAM
- Об'ємне захоплення (volumetric capture) - реконструкція тривимірної моделі сцени на основі RGB-D-даних з передачею дистанційному учаснику

3. За моделлю взаємодії між учасниками:

- Асиметрична (expert-guided): виконавець транслює відео, оператор-експерт накладає вказівки
- Симетрична (collaborative): обидва учасники бачать спільний простір і можуть взаємодіяти з об'єктами
- Змішана: поєднання обох режимів залежно від стану сесії

4. За архітектурою транспортного рівня:

- Peer-to-peer (WebRTC або власні UDP-протоколи)
- Клієнт-серверна (відеопотік маршрутизується через хмарний сервер)
- Гібридна (LAN-режим для локальних мереж з хмарним резервуванням)

5. За режимом розгортання:

- SaaS (хмарне рішення провайдера)
- On-premises (розгортання у корпоративній мережі)
- Edge/локальне (обробка безпосередньо на пристрої або у локальній мережі)

Аналізуючи сучасний стан ринку, варто розглянути провідні комерційні рішення. На сьогодні ринок комерційного програмного забезпечення для AR-підтримки представлений рядом зрілих платформ. Нижче розглянуті найбільш поширені рішення.

Microsoft Dynamics 365 Remote Assist є флагманським корпоративним рішенням компанії Microsoft, орієнтованим на взаємодію з гарнітурою HoloLens 2, а також мобільними пристроями на Android та iOS. [4] Система інтегрується з Microsoft Teams, що дозволяє дистанційному учаснику без спеціалізованого обладнання підключатися до сесії підтримки через стандартний браузер або настільний клієнт. Серед функцій - передача AR-анотацій, завантаження PDF-документів і зображень у спільний простір, групові дзвінки, а також тісна інтеграція з Dynamics 365 Field Service для управління нарядами. Система реалізує просторово прив'язані анотації у world-space, які зберігають своє положення відносно об'єктів реального світу завдяки трекінговим можливостям HoloLens. Суттєвим обмеженням є жорстка прив'язка до екосистеми Microsoft Azure та необхідність у ліцензіях Azure Active Directory, що ускладнює застосування у корпоративних середовищах сторонніх організацій.

TeamViewer Assist AR (раніше - TeamViewer Pilot) [5] реалізує підхід «see-what-I-see», за якого дистанційний експерт отримує відеопотік з камери виконавця і може накладати анотації на зображення в реальному часі. Рішення позиціонується як кросплатформне і підтримує смартфони, планшети, а також промислові смарт-

окуляри RealWear, Vuzix тощо. TeamViewer Assist AR орієнтований на класичну screen-space модель анотацій, де позначки не прив'язані до тривимірної геометрії сцени, - це спрощує реалізацію, проте знижує просторову точність при зміні кута огляду або переміщенні виконавця.

PTC Vuforia Chalk - рішення компанії PTC, що поєднує AR-анотації з функціями трекінгу на основі технологій Vuforia. Система дозволяє накладати «живі» позначки на об'єкти у кадрі, які автоматично адаптуються при переміщенні пристрою виконавця. Vuforia Chalk є частиною ширшої платформи промислової AR від PTC і може інтегруватися з системами PLM/ERP. Основним обмеженням залишається залежність від хмарної інфраструктури PTC та відносно висока вартість корпоративного ліцензування.

Scope AR WorkLink позиціонується як корпоративна платформа для підготовки і надання AR-інструкцій із підтримкою гарнітур HoloLens, Google Glass та мобільних пристроїв. На відміну від перерахованих рішень, Scope AR акцентує увагу на заздалегідь підготовленому AR-контенті (покрокові інструкції), а не на сесіях реального часу.

Узагальнена характеристика розглянутих комерційних систем наведена у табл. 1.1. [4-6]

Таблиця 1.1 - Порівняльна характеристика комерційних систем AR-підтримки

Система	Пристрій виконавця	Тип анотацій	Транспорт	Розгортання
MS Dynamics 365 Remote Assist	HoloLens 2, Android, iOS	World-space (HoloLens), Screen-space (mobile)	WebRTC / Teams	SaaS (Azure)
TeamViewer Assist AR	Смартфон, смарт-окуляри	Screen-space	WebRTC	SaaS / On-premises

Система	Пристрій виконавця	Тип анотацій	Транспорт	Розгортання
PTC Vuforia Chalk	Смартфон, планшет	Screen-space + object tracking	Хмарний	SaaS
Scope AR WorkLink	HoloLens, Google Glass, mobile	World-space	Хмарний	SaaS / On-premises

Паралельно з комерційними рішеннями у науковій літературі описано ряд дослідницьких прототипів, що демонструють перспективні архітектурні підходи. Система ARRA [2], розроблена у Cranfield University, реалізує маніпуляцію віртуальними об'єктами дистанційним оператором з накладенням результатів на реальне оточення виконавця, що було верифіковано в експерименті з 60 учасниками. Система для дистанційної підтримки під час проведення серцево-легеневої реанімації, описана у праці «CPR Emergency Assistance Through Mixed Reality Communication», використовує RGB-D-камери для об'ємного захоплення сцени і передачі просторово точного представлення дистанційному реаніматологу. Такий підхід забезпечує найвищу точність просторового розуміння, проте вимагає значних обчислювальних ресурсів і спеціалізованого апаратного забезпечення, що обмежує його практичну застосовність.

Огляд AR у реальному часі телемедицини [3], що охоплює 20 платформ і пристроїв, виявив, що спільними функціями для більшості рішень є можливість дистанційного анотування, відображення графічних елементів, а також передача зображення рук або інструментів у поле зору місцевого користувача. При цьому дослідники відзначають, що ранній стан технічного розвитку галузі та відсутність стандартизованих інструментів стримують широкомасштабне впровадження AR-підтримки у чутливих операційних середовищах.

Аналіз наявних систем дозволяє виокремити ряд характерних обмежень, що визначають актуальність розробки системи V.I.R.A.

По-перше, переважна більшість комерційних рішень орієнтована на гарнітури класу optical see-through (HoloLens, Magic Leap) або смартфони з

ARCore/ARKit, тоді як потенціал гарнітур змішаної реальності з повнокольоровим відеопрозорим (video passthrough) дисплеєм, зокрема Meta Quest 3S, залишається значною мірою нереалізованим у контексті дистанційної підтримки.

По-друге, більшість систем функціонує виключно через хмарну інфраструктуру провайдера без підтримки локального (LAN) режиму. Це унеможливлює їх застосування в умовах обмеженого або закритого інтернет-доступу, що є типовим для промислових об'єктів, підземних споруд або захищених корпоративних мереж.

По-третє, жодне з розглянутих рішень не реалізує власний UDP-протокол для стримінгу відео з гарнітури, з окремим транспортним каналом для анотацій. Натомість усі вони покладаються або на WebRTC (де весь потік маршрутизується через TURN-сервери), або на фірмові хмарні канали. Такий підхід ускладнює оптимізацію затримки для конкретних умов мережі.

По-четверте, системи з комерційним SaaS-розгортанням не забезпечують повного контролю над обробкою відеоданих і метаданих анотацій, що є суттєвим обмеженням для застосувань з підвищеними вимогами до конфіденційності, зокрема в медичній, оборонній або атомній галузях.

Система V.I.R.A. проектується як відповідь на перелічені обмеження: вона реалізує власний фрагментований UDP-протокол для відео, підтримує двошляхову транспортну архітектуру (LAN та хмарний режим), забезпечує world-space прив'язку анотацій на гарнітурі Meta Quest 3S та передбачає повне self-hosted розгортання серверної частини.

1.2. Технології змішаної реальності та passthrough-архітектура гарнітур

Концептуальною основою для розуміння технологій змішаної реальності слугує концепція «континууму реальність-віртуальність» (Reality-Virtuality Continuum), запропонована Полом Мілгремом і Фуміо Кісіно у 1994 році. [7]

Відповідно до цієї моделі, між двома полюсами - повністю фізичним оточенням і повністю синтетичним віртуальним середовищем - розташована широка область змішаної реальності (Mixed Reality, MR), яка охоплює як доповнену реальність (Augmented Reality, AR), так і доповнену віртуальність (Augmented Virtuality, AV). У доповненій реальності переважає реальний світ, доповнений комп'ютерно-генерованими елементами, тоді як у доповненій віртуальності домінує синтетичне середовище, у яке вбудовуються реальні об'єкти.

Сучасна термінологія дещо розширює початкове визначення: під «змішаною реальністю» нині часто розуміють інтерактивні системи, у яких цифровий і фізичний простори взаємодіють одночасно, а не просто накладаються один на одного. Саме цей ширший зміст закладено у назву пристроїв класу Meta Quest 3/3S, Apple Vision Pro та Microsoft HoloLens 2, що позиціонуються виробниками як MR-гарнітури.

Класифікація пристроїв HMD за оптичною архітектурою є головним критерієм апаратної оцінки гарнітур з можливостями AR/MR. За цим критерієм виділяють два принципові підходи:

Перший підхід — *Optical See-Through (OST)*, що базується на принципі оптичної прозорості. Гарнітура оснащена напівпрозорими комбайнерами або хвилеводними дисплеями (waveguide), через які користувач безпосередньо бачить фізичне оточення, тоді як проектований синтетичний зміст накладається зверху. Типові представники - Microsoft HoloLens 2 та Magic Leap 2 [8, 9]. Принципова перевага підходу полягає у відсутності затримки передачі реального зображення та збереженні природного сприйняття динамічного діапазону оточення. Серед недоліків - вузьке поле зору (FOV) дисплея (зазвичай 40-70°), нездатність забезпечити оклюзію реальних об'єктів синтетичними, а також значна складність і висока вартість виробництва хвилеводних пластин.

Другий підхід — *Video See-Through (VST)* або *Passthrough* (відеопрозорість). У цьому випадку гарнітура повністю блокує пряме поле зору і формує зображення фізичного оточення з масиву вбудованих зовнішніх камер, відтворюючи його на внутрішніх дисплеях у реальному часі. Синтетичні елементи рендеряться разом із

відтвореним відеопотоком, забезпечуючи повну піксельну поверхню для MR-сцени. Серед переваг - широке FOV, можливість оклюзії, відсутність обмежень хвилеводних оптичних елементів та нижча собівартість. Суттєвим недоліком є неминуча затримка passthrough-конвеєра (photon-to-photon latency), яка вносить відчутне запізнення між фізичним рухом і його відображенням на екрані гарнітури. Порівняльну характеристику двох підходів наведено у табл. 1.2.

Таблиця 1.2 - Порівняльна характеристика оптичних архітектур гарнітур змішаної реальності

Критерій	OST (HoloLens 2, Magic Leap)	VST/Passthrough (Meta Quest 3/3S)
Затримка реального світу	~0 ms (оптична)	35-40 ms (Quest 3)
FOV синтетичного зображення	43-70° (обмежено дисплеєм)	~96-110° (весь дисплей)
Оклюзія реал. об'єктів	Лише часткова / відсутня	Можлива при наявності depth-sensor
Природне сприйняття кольору	Так	Залежить від якості камер
Вартість	Висока	Значно нижча
Придатність для стримінгу відео	Складна (немає відеопотоку оточення)	Висока (є повний відеопотік)

Детального розгляду потребує passthrough-архітектура гарнітури Meta Quest 3S. Це пристрій автономного класу, що базується на процесорі Qualcomm Snapdragon XR2 Gen 2 з 8 ГБ оперативної пам'яті та забезпечує MR-функціональність через повнокольоровий passthrough. Апаратна база passthrough-підсистеми включає два RGB-кольорових сенсори для стереоскопічного захоплення фізичного оточення з роздільністю 18 пікселів на градус (PPD) - показник, що у 4,5 рази перевищує якість монохромного passthrough попереднього покоління (Quest 2).

Архітектурно passthrough-конвеєр функціонує наступним чином. Пара RGB-камер синхронно захоплює кадри фізичного оточення. Кожний кадр передається

GPU для корекції дисторсії об'єктивів, перепроекції у систему координат очей (view-space reprojection) з урахуванням міжзіничної відстані та поточної орієнтації гарнітури, а також колірної нормалізації. Результуюче зображення подається на LCD-панелі гарнітури. Оскільки камери фізично рознесені відносно оптичних центрів лінз, реальне зображення завжди проходить через цей конвеєр незалежно від наявності синтетичного контенту.

Документація Meta Passthrough Camera API [10] вказує такі характеристики підсистеми захоплення: затримка захоплення зображення (image capture latency) - 20-40 мс, частота оновлення - 60 Гц, максимальна роздільність кадру - 1280×1280 пікселів, внутрішній формат даних - YUV420. Виміряна незалежними дослідниками (Optofidelity) [11] повна затримка passthrough (photon-to-photon latency) для Meta Quest 3 становить близько 39 мс, що є типовим значенням для VST-гарнітур цього покоління і залишається суттєво більшим, ніж 11 мс для Apple Vision Pro. Водночас для задач дистанційної підтримки, де вся система вже оперує мережевою затримкою у десятки мілісекунд, дана величина не є визначальним обмеженням.

Просторова прив'язка синтетичних об'єктів до фізичного оточення є фундаментальною вимогою будь-якої AR/MR-системи. Сучасні автономні гарнітури, включаючи Meta Quest 3S, реалізують принцип **inside-out трекінгу** на основі візуально-інерціального одночасного визначення місцезнаходження та побудови карти (Visual-Inertial SLAM, VI-SLAM) [13]. На відміну від зовнішніх систем трекінгу (outside-in, наприклад Lighthouse від HTC Vive), inside-out підхід не потребує розгортання стаціонарної інфраструктури і є самодостатнім.

VI-SLAM поєднує два джерела сенсорних даних: монохромні трекінгові камери, що надають структурну (ключові точки) та оптичну (оптичний потік) інформацію про оточення, та IMU (інерціальний вимірювальний пристрій), що забезпечує швидкі попередні оцінки між кадрами камер. Алгоритм безперервно будує та уточнює внутрішню тривимірну карту оточення (feature map або voxel map), у системі координат якої визначається 6-DoF поза гарнітури (три координати положення + три кути орієнтації). Саме ця карта є основою для *просторових якорів*

(spatial anchors) - механізму, що дозволяє прив'язувати Unity-об'єкти до конкретних точок реального простору таким чином, щоб вони залишалися нерухомими при переміщенні користувача.

У системі V.I.R.A. просторовий трекінг Meta Quest 3S використовується як основа для прив'язки AR-анотацій і семантичних маркерів у world-space - детальніше це розглядається у підрозділах 2.4 та 2.2. Критично важливою властивістю є те, що VI-SLAM надає стійкий фрейм координат, доступний через API OpenXR безпосередньо у Unity-застосунку, без необхідності в зовнішніх маркерах або інфраструктурі.

У Unity-застосунку для Meta Quest 3S MR-рендеринг реалізується через шар passthrough як фоновий шар OpenXR Passthrough Layer, поверх якого рендеряться синтетичні об'єкти з підтримкою альфа-композитингу. Meta XR SDK надає компонент OVRPassthroughLayer, що керує параметрами passthrough-рендерингу: прозорістю, контрастом, підмішуванням кольору та інтенсивністю вставки синтетичних об'єктів. З точки зору конвеєру рендерингу, кожний кадр формується у два проходи: спочатку passthrough-фрейм від камер через OpenXR-шар, потім - фіналізована сцена зі всіма синтетичними об'єктами і анотаціями, складена через апаратний compositing GPU.

Характерна особливість Meta Quest 3S, що відрізняє її від OST-гарнітур у контексті системи V.I.R.A., - наявність повного RGB-відеопотоку з камер, доступного не лише для відображення passthrough, але й для зовнішньої трансляції. Саме цей відеопотік є джерелом для UDP-стримінгу до Android Relay та далі до браузерного інтерфейсу оператора. Таким чином, passthrough-підсистема виконує у V.I.R.A. подвійну функцію: забезпечення MR-середовища для виконавця на місці та формування відеоматеріалу для дистанційного оператора.

1.3. Протоколи реального часу: UDP, WebSocket та WebRTC

Аналізуючи транспортний рівень і вимоги реального часу, слід зазначити, що будь-яка система дистанційної підтримки функціонує в умовах жорстких часових обмежень. Вимога до наскрізної затримки (end-to-end latency) в інтерактивних системах телеприсутності зазвичай визначається порогом 150 мс [16] - значенням, вище якого учасники сесії відчувають помітні затримки у взаємодії. Для відеопотоку у режимі спостереження допустимим вважається значення до 250-400 мс, тоді як для голосу та синхронізованих анотацій - менше 100 мс. Вибір транспортного протоколу є ключовим архітектурним рішенням, що безпосередньо визначає досяжність цих показників.

Сучасний стек протоколів для систем реального часу охоплює три принципово різні рівні абстракції: *UDP* як базовий дейтаграмний транспорт, *WebSocket* як протокол повнодуплексного з'єднання поверх TCP, та *WebRTC* як стандартизований набір протоколів і API для peer-to-peer медіакомунікацій у браузерах. Кожен із них вирішує різні задачі і має чітко визначену область застосування, що й обумовлює їх спільне використання у системі V.I.R.A.

Протокол UDP та його властивості для стримінгу мають вирішальне значення. UDP (User Datagram Protocol) є дейтаграмним транспортом, визначеним у RFC 768 (1980). На відміну від TCP, UDP не встановлює з'єднання, не гарантує доставки пакетів, не контролює їх порядок і не забезпечує механізму повторної передачі втрачених сегментів. Саме ці властивості, що формально виглядають як недоліки, є вирішальними перевагами для передачі медіапотоків реального часу.

Фізичний аналіз поведінки двох протоколів під час передачі відео наочно ілюструє різницю підходів. У разі втрати пакету TCP ініціює механізм повторної передачі (retransmission), що призводить до блокування всього потоку даних (head-of-line blocking) на час RTT (Round-Trip Time). Для відеостримінгу це означає накопичення буфера та стрибкоподібні затримки. UDP натомість просто відкидає втрачений пакет: UDP-потік може втратити 0.1% пакетів із ледве помітною

деградацією якості зображення, тоді як TCP у тій самій ситуації може ввести кількасекундну затримку буферизації.

Практичне скло-до-скла (glass-to-glass) відео-затримку за різними протоколами ілюструє наступне порівняння: HLS - 10-30 с, RTMP - 3-5 с, SRT - 1-3 с, WebRTC (UDP) - 0.5-2 с [19, 20]. Саме WebRTC над UDP є еталоном мінімальної затримки для браузерних відеосистем, проте при безпосередній реалізації на гарнітурі або мобільному пристрої «чистий» UDP без надбудови WebRTC забезпечує ще нижчу затримку, оскільки усуває накладні витрати ICE-узгодження та DTLS-шифрування.

Проте UDP вимагає реалізації прикладного рівня для вирішення задач, які у TCP виконуються автоматично: фрагментації великих повідомлень (оскільки стандартний MTU Ethernet дорівнює 1500 байт, а з урахуванням заголовків IP і UDP - 1472 байти корисного навантаження), ідентифікації фреймів, збирання фрагментів та виявлення застарілих буферів. У системі V.I.R.A. максимальний розмір UDP-пакету обмежено значенням MAX_UDP_PACKET_SIZE = 60000 байт (нижче теоретичного ліміту датаграми ~65507 байт) - цим досягається баланс між кількістю фрагментів і імовірністю фрагментації на мережевому рівні.

Окрему роль відіграє *WebSocket* — повнодуплексний канал над TCP. Цей протокол прикладного рівня, стандартизований у RFC 6455 (2011) [15], який забезпечує повнодуплексний канал зв'язку поверх єдиного TCP-з'єднання. Протокол вирішує фундаментальне обмеження HTTP: класичний запит-відповідь є однонаправленим і не підходить для серверних push-повідомлень без використання опитування (polling) або long-polling - підходів, що породжують надлишкові HTTP-заголовки та множинні TCP-з'єднання на кожного клієнта.

WebSocket-з'єднання ініціюється стандартним HTTP-запитом з оновленням (Upgrade handshake), після чого перемикається у режим повнодуплексного фреймованого потоку. Фрейми WebSocket мають мінімальний заголовок (2-14 байт), підтримують бінарні та текстові дані, а також вбудований механізм ping/pong для контролю живучості з'єднання. Протокол нативно підтримується браузерами та

не потребує плагінів, що робить його стандартом де-факто для браузерних застосунків реального часу.

Принципове обмеження WebSocket у контексті медіапотоків - це успадкована від TCP гарантія доставки та упорядкованості. При значних втратах пакетів або congestion у мережі TCP-буферизація може призводити до тих самих ефектів head-of-line blocking, що й при HTTP-стримінгу. Для передачі текстових команд і керуючих повідомлень (анотацій, станів, сигналів) цей механізм є бажаним - жодна команда не повинна бути втрачена. Для відеопотоку він субоптимальний, але прийнятний у хмарному сегменті мережі, де якість каналу між Android Relay та хмарним сервером зазвичай вища і стабільніша, ніж у локальній Wi-Fi-мережі між гарнітурою та телефоном.

У системі V.I.R.A. WebSocket виконує роль транспорту у двох вузлах: між Android Relay та хмарним Node.js-сервером (бінарні JPEG-фрейми + текстові JSON-команди в одному з'єднанні) та між хмарним сервером і браузерним клієнтом оператора (WebSocket-relay для відеопотоку у LAN-режимі та сигналізація WebRTC у хмарному режимі).

Протокол *WebRTC*, його архітектура та стек, являє собою відкритий стандарт для peer-to-peer медіакомунікацій. [21, 22]. Зокрема аудіо-, відео- та довільного потоку даних між браузерами (або іншими WebRTC-сумісними кінцевими точками) без проміжних серверів. Стандарт охоплює кілька шарів, кожен з яких вирішує окрему задачу.

На транспортному рівні медіадані WebRTC передаються протоколом RTP поверх UDP. Медіадані WebRTC передаються протоколом RTP (Real-time Transport Protocol) поверх UDP, що забезпечує низьку затримку і допускає втрати пакетів. Безпека забезпечується DTLS (Datagram Transport Layer Security) для узгодження ключів і SRTP (Secure RTP) для шифрування медіапотоку. Для потоків даних використовується SCTP поверх DTLS.

Застосування механізмів ICE, STUN та TURN є відповіддю на виклик NAT-traversal. Одним із ключових викликів WebRTC є NAT-traversal: пристрої в окремих приватних мережах не мають прямої маршрутизації між собою. Механізм ICE

(Interactive Connectivity Establishment, RFC 8445) [17, 18] вирішує цю задачу через послідовне тестування кандидатних адрес трьох типів: локальні адреси, server-reflexive адреси (отримані через STUN-сервер, RFC 5389, що відображає зовнішній IP клієнта) та relay-адреси (через TURN-сервер, RFC 5766, що ретранслює трафік у разі відсутності прямого з'єднання). Процес ICE-узгодження зазвичай завершується за десятки мілісекунд.

Процес сигналізації у WebRTC стандартно не специфікується, що дозволяє розробникам вибирати довільний зовнішній канал. WebRTC стандартно не специфікує протокол сигналізації - він відповідає лише за медіатранспорт. Для обміну SDP (Session Description Protocol) - описами кодеків і можливостей сесії - та ICE-кандидатами між рівноправними вузлами потрібен довільний зовнішній канал. У переважній більшості реалізацій для цього використовується WebSocket-з'єднання до сигнального сервера. Цей підхід реалізований і у V.I.R.A.: хмарний Node.js-сервер виступає WebRTC-сигнальним сервером, який маршрутизує SDP offer/answer та ICE-кандидати між браузерним оператором та Android Relay.

Архітектурно процес встановлення WebRTC-сесії відбувається у такій послідовності:

1. Обидва учасники реєструються на сигнальному сервері (повідомлення webrtc:register)
2. Ініціатор (браузер оператора) формує RTCPeerConnection і через STUN отримує своїх ICE-кандидатів
3. Ініціатор надсилає SDP offer через WebSocket-сигналізацію
4. Другий учасник (Android Relay або Quest) відповідає SDP answer
5. Обидва учасники обмінюються ICE-кандидатами у режимі trickle ICE
6. ICE вибирає оптимальний шлях (прямий P2P або relay через coturn) і встановлює DTLS-з'єднання
7. Після DTLS-рукостискання розпочинається передача медіаданих по SRTP/SCTP

У V.I.R.A. TURN-сервер реалізований на базі програмного забезпечення *coturn*, розгорнутого на хмарному вузлі Hetzner CAX11 разом із Node.js-сервером.

coturn забезпечує UDP/TCP relay-трафік для ситуацій симетричного NAT, коли пряме peer-to-peer з'єднання між браузером і Android Relay неможливе.

Три розглянутих протоколи вирішують у системі V.I.R.A. різні транспортні задачі. Їх властивості у контексті системи зведено у табл. 1.3.

Таблиця 1.3 - Порівняльна характеристика протоколів реального часу

Властивість	UDP (Quest → Relay)	WebSocket (Relay → Server → Browser)	WebRTC (Browser ↔Relay)
Транспорт	UDP датаграми	TCP (повнодуплекс)	UDP/RTP (DTLS)
Гарантія доставки	Відсутня	Так (TCP)	Відсутня (RTP), часткова NACK
Порядок пакетів	Не гарантований	Гарантований	Не гарантований (RTP seq)
Затримка	Мінімальна (~5-20 мс LAN)	Середня (залежить від TCP congestion)	Низька (0.5-2 с glass-to-glass)
NAT traversal	Потребує Relay (Android)	Ні (вихідне з'єднання)	ICE + STUN/TURN (coturn)
Шифрування	Відсутнє (мережа LAN)	TLS (WSS у хмарі)	DTLS + SRTP (mandatory)
Призначення у V.I.R.A.	Відеофрагменти + анотації	Хмарний relay відео + сигналізація	Медіапотік оператора (хмарний режим)

Двошарова транспортна архітектура V.I.R.A. (LAN-режим через UDP+WebSocket та хмарний режим через WebRTC) є прямим наслідком відсутності єдиного протоколу, який оптимально задовольняє всі вимоги системи одночасно: мінімальну затримку, NAT-traversal, браузерну сумісність і можливість роботи у закритих локальних мережах. Детальна реалізація цієї архітектури розглядається у підрозділах 2.5, 2.6 та 2.8.

1.4. Методи просторового прив'язування та анотування у середовищі MR

Архітектура системи насамперед передбачає вирішення ключової задачі просторової реєстрації (spatial registration) - точному суміщенні цифрового об'єкта з конкретною точкою або поверхнею реального фізичного простору таким чином, щоб цей об'єкт зберігав своє положення при переміщенні користувача. Якість реєстрації визначається двома взаємопов'язаними вимірами: *точністю* (accuracy) - наскільки близько цифровий елемент знаходиться до цільової точки у просторі - та *стабільністю* (stability, world-locking accuracy) - наскільки елемент «прилипає» до свого місця при русі голови і накопиченні дрейфу трекінгу з часом. Відхилення реєстрації навіть у кілька сантиметрів руйнує когнітивне відчуття прив'язки анотації до реального об'єкта, що підтверджується дослідженнями у галузі AR-дистанційної підтримки.

У системах дистанційної підтримки задача ускладнюється асиметричністю взаємодії: дистанційний оператор бачить лише двовимірний відеопотік і задає позицію анотації в екранних координатах, тоді як виконавець на місці повинен отримати анотацію, прив'язану у тривимірному просторі. Ця трансформація - від 2D screen-point до 3D world-point - є ключовою обчислювальною операцією, що визначає якість просторового анотування і для реалізації якої необхідно вирішити проблему визначення глибини сцени.

Класифікація підходів до анотування дозволяє виділити два основні типи графічних елементів:

Перший тип — *screen-space* анотації (екранні, або 2D-анотації), що формуються безпосередньо у площині дисплея і зберігаються у системі координат екрану чи відеофрейму. Вони не мають прив'язки до тривимірної геометрії сцени: при переміщенні або повороті головою позначка «ковзає» разом із зображенням або залишається нерухомою у куті поля зору. Цей підхід технічно найпростіший - не потребує depth-сенсора, SLAM-карти або тривимірних розрахунків - і широко застосовується у комерційних системах першого покоління, зокрема TeamViewer

Assist AR та PTC Vuforia Chalk. Головний недолік полягає у відсутності просторового сенсу: при зміні кута огляду анотація більше не вказує на правильний об'єкт, а при переміщенні виконавця - «пливе» разом із картинкою замість того, щоб залишатися на реальній поверхні.

Другий тип — *world-space анотації* (просторові, або 3D-анотації), які прив’язуються до координат тривимірного простору і стабільно залишаються там незалежно від руху користувача. Для їх реалізації необхідні: тривимірна поза камери у момент розміщення, метод визначення глибини для перетворення 2D→3D та механізм збереження прив'язки у просторовій карті середовища. Дослідження підтверджують, що world-space анотації суттєво підвищують ефективність просторової комунікації: у порівняльних роботах умови world-space приводили до зниження кількості помилок виконавця та скорочення часу виконання завдань порівняно зі screen-space умовами [23, 24].

Окрім вільних штрихових анотацій (strokes), у сучасних MR-системах підтримки застосовуються *семантичні маркери* - стандартизовані іконографічні символи з типізованим змістом: «Увага», «Точка вимірювання», «Місце кріплення» тощо [26]. На відміну від вільних ліній, маркер є самостійним семантичним об'єктом, що несе однозначне повідомлення без необхідності в усному поясненні. Дослідження CoCreatAR (2025) [25] підтверджують, що семантично типізовані анотації знижують когнітивне навантаження на виконавця і зменшують неоднозначність у трактуванні вказівок порівняно з вільними позначками.

Технологія VI-SLAM виступає фундаментальною основою просторової прив'язки об’єктів. Передумовою реалізації world-space анотацій є наявність надійного механізму трекінгу положення гарнітури у просторі. Сучасним стандартом для автономних MR-гарнітур є візуально-інерціальна SLAM (Visual-Inertial Simultaneous Localization and Mapping, VI-SLAM) - алгоритм, що поєднує дані монохромних трекінгових камер з даними IMU (інерціального вимірювального пристрою) для безперервної оцінки 6-DoF пози гарнітури та побудови внутрішньої карти середовища.

VI-SLAM вирішує класичну задачу «курки і яйця»: для локалізації потрібна карта, а для побудови карти - локалізація. Розв'язання досягається через спільну оптимізацію траєкторії та позицій ключових точок (features) у карті, зазвичай формульовану як задача нелінійної найменших квадратів або factor-graph оптимізації. Компонент IMU відіграє критичну роль у двох аспектах: по-перше, надає швидкі оцінки між кадрами камер (high-rate pose prediction), по-друге, визначає метричний масштаб сцени - проблему, нерозв'язну для моновізуального SLAM без ІНС або відомих маркерів. Точність сучасних VI-SLAM систем, зокрема ORB-SLAM3 [27, 28], досягає 9 мм при типових AR-рухах у приміщенні на датасеті TUM-VI.

Практично важливою властивістю VI-SLAM для AR-застосунків є поняття *drift* (дрейф) - повільне накопичення похибки поз, що виникає через помилки у вимірюваннях сенсорів, обмежений час витримки камер та зміну умов освітлення. Дрейф є принципово невідворотнім у системах без абсолютного позиційного сигналу (GPS, ультразвукові маяки) і може призводити до поступового «зісковзування» прив'язаних об'єктів з їхніх позицій протягом тривалих сесій. Для компенсації дрейфу у VI-SLAM використовуються: *loop closure* - виявлення повторного відвідування вже картографованих місць і глобальна оптимізація графа поз, та *relocalization* - відновлення пози після втрати трекінгу.

Аналіз методів визначення глибини для перетворення 2D→3D виявляє кілька основних підходів. Ключовим компонентом реалізації world-space анотацій є перетворення нормалізованих екранних координат $(u, v) \in [0,1]^2$, отриманих від оператора, у тривимірну точку світового простору ($\mathrm{p} \in \mathbb{R}^3$) . Це перетворення потребує побудови промінчика (ray) із центру проекції камери крізь точку (u, v) на зображенні та визначення відстані вздовж цього промінчика до реальної поверхні.

Існує кілька фізично відмінних методів отримання глибини:

Методи структурованого світла (structured light) та ToF використовують активне випромінювання. Активні depth-сенсори проектують відомий інфрачервоний патерн або вимірюють час прольоту імпульсу, і за результатом формують пікселізовану карту глибини. Метод забезпечує щільну і точну карту, але

чутливий до зовнішнього освітлення та дистанції. Типові представники - Microsoft Azure Kinect (ToF), Intel RealSense D435 (structured light + stereo). Meta Quest 3/3S використовує аналогічний принцип у своєму Environment Depth API, надаючи GPU-текстуру глибини для raycast-операцій.

Стереоскопічна пасивна тріангуляція базується на порівнянні двох зміщених зображень. Дві рознесені камери формують стереопару, а з диспаратності відповідних пікселів геометрично обчислюється глибина. Метод не потребує проектора, але чутливий до текстури поверхні (у безтекстурних регіонах дисперсія глибини зростає) та до точності калібровки. Є стандартним підходом для мобільних пристроїв без спеціалізованого depth-сенсора.

Монокулярна оцінка глибини (monocular depth estimation) залучає нейронні мережі для прогнозування карти глибини. Глибина виводиться з одного зображення на основі навчених нейромережевих моделей, що використовують сигнали перспективи, розмиття, оклюзії та семантики сцени. Підхід не потребує додаткового обладнання, але дає відносну (unnormalized) глибину і не гарантує метричної точності. Застосовується у системах, де немає depth-сенсора і неможлива стереокалібровка.

Метод фіксованої глибини (fixed-depth fallback) полягає у розміщенні анотацій на заданій відстані від камери. Найпростіший підхід: анотація розміщується на константній відстані d_0 від камери вздовж вектора промінчика незалежно від реальної геометрії сцени. Є грубим наближенням, придатним для прототипування або як запасний варіант при відмові depth-сенсора. Точність визначається тим, наскільки реальна глибина сцени відрізняється від константи d_0. Порівняльну характеристику методів наведено у табл. 1.4.

Таблиця 1.4 - Характеристика методів визначення глибини сцени

Метод	Обладнання	Щільність карти	Метрична точність	Обмеження
ToF / Structured light	Спец. depth-сенсор	Щільна	Висока (~мм)	ІЧ-перешкоди, дистанція

Метод	Обладнання	Щільність карти	Метрична точність	Обмеження
Стерео тріангуляція	Стереокамера	Щільна	Середня	Безтекстурні поверхні
Моно нейромережева	Будь-яка камера	Щільна	Відносна	Немає метричного масштабу
Фіксована глибина	Будь-яка камера	-	Груба (±залежно від сцени)	Помилки при глибині ≠ d_0

Важливим інструментом стабілізації є просторові якорі, їх концепція та механізм програмної реалізації. *Просторовий якір* (spatial anchor) є стандартизованою абстракцією у сучасних MR-платформах для стабільної довготривалої прив'язки цифрових об'єктів до реального простору. Microsoft Mixed Reality визначає просторовий якір [29] як «важливу точку у просторі, яку система відстежує з часом; кожен якір має власну систему координат, скориговану відносно інших якорів або систем відліку, щоб прив'язані голограми залишалися на місці». Принципова відмінність якоря від звичайного об'єкта сцени з фіксованими координатами полягає в тому, що SDK автоматично коригує трансформацію якоря при кожному оновленні SLAM-карти, компенсуючи накопичений дрейф трекінгу.

Meta рекомендує розміщувати прив'язані контентні об'єкти у межах 3 метрів від початку координат якоря: при більшій відстані виникають похибки через «lever-arm effect» - ефект важеля, коли мала похибка орієнтації якоря (наприклад, 0.5°) трансформується у значне лінійне зміщення далекого об'єкта (наприклад, ~8 см на відстані 10 м). З цієї причини архітектурною рекомендацією є призначення окремого якоря для кожної анотації у точці її розміщення, а не використання одного спільного якоря для всієї сцени.

Окремим питанням є персистентність якорів між сесіями: якщо анотації повинні зберігатися після вимкнення гарнітури і знову з'являтися при повторному запуску, якорі потрібно зберігати локально або в хмарному просторовому сховищі (Shared Spatial Anchors, Cloud Spatial Anchors). Поточні можливості платформ

дозволяють зберігати якорі для одного пристрою локально або синхронізувати їх між кількома пристроями через хмарні сервіси Meta або Azure, хоча крос-платформна сумісність якорів між різними виробниками гарнітур досі є відкритою дослідницькою задачею.

Окремої уваги заслуговує стабільність world-space об'єктів при використанні passthrough-режиму. Практичним викликом при спільному використанні просторових якорів і passthrough-режиму є явище *«wobbling»* - мікровібрація прив'язаних об'єктів, що виникає через десинхронізацію між конвеєром відеопрозорості (passthrough pipeline, затримка ~39 мс) та конвеєром відстеження поз (tracking pipeline, затримка <1 мс). Оскільки passthrough-зображення завжди відображає світ з запізненням, а синтетичні об'єкти рендеряться із мінімальною затримкою трекінгу, при швидких рухах голови між ними виникає видима часова неузгодженість. Ця проблема є фундаментальною для всіх VST-гарнітур і не може бути повністю усунена без апаратного зменшення passthrough-латентності. Практичний мінімум на сьогоднішній день становить ~11 мс (Apple Vision Pro) проти ~39 мс (Meta Quest 3).

Сукупність розглянутих методів - VI-SLAM-трекінг, depth raycast для 2D→3D перетворення, OVRSpatialAnchor як механізм стабілізації та семантичні маркери як форма типізованих анотацій - утворює технологічну основу для реалізації просторового анотування у системі V.I.R.A. Конкретні архітектурні рішення, прийняті при розробці відповідних компонентів, та їх обґрунтування детально розглядаються у підрозділі 2.4.

1.5. Клієнт-серверні архітектури для потокового відео

Загальна модель відеостримінгового конвеєра в MR-системах включає кілька послідовних етапів. Сучасний відеостримінговий конвеєр є ланцюжком перетворень сигналу від джерела зображення до дисплею кінцевого споживача, де

кожна ланка вносить свій внесок у сукупну затримку, якість і стійкість потоку. Огляд «An End-to-End Pipeline Perspective on Video Streaming» [30] виділяє у цьому ланцюжку такі ключові стадії: захоплення (capture), кодування (encoding), упакування та сегментація (packaging), доставка через мережу (delivery), буферизація та декодування на клієнті (buffering, decoding), і нарешті відображення (rendering). Кожна з цих стадій може бути виконана на різних обчислювальних вузлах - від пристрою-джерела до хмарного сервера або граничного вузла - і вибір архітектурної топології принципово впливає на баланс між затримкою, масштабованістю та обчислювальними витратами.

Для інтерактивних систем дистанційної підтримки архітектурні вимоги суттєво відрізняються від масових відеосервісів (YouTube, Netflix): домінуючим критерієм є мінімізація наскрізної затримки (glass-to-glass latency), тоді як масштабованість на тисячі одночасних глядачів і адаптивний бітрейт відходять на другий план. Ця принципова відмінність визначає вибір протоколів і топологій, розглянутих у даному підрозділі.

Розгляд протоколів доставки відео, їх класифікація та аналіз затримок дозволяють зробити такі висновки:

Різноманіття протоколів стримінгу відео зручно систематизувати за двома осями: базовий транспорт (TCP або UDP) та наскрізна затримка. Від цих параметрів похідними є масштабованість, адаптивність і браузерна сумісність.

Протоколи HLS та MPEG-DASH є сегментованими і забезпечують високу стабільність. Це сегментовані HTTP-протоколи, де відеопотік розбивається на фрагменти фіксованої тривалості (зазвичай 2-10 с), які доставляються окремими HTTP-запитами. Типова затримка - 6-30 с. Ці протоколи забезпечують видатну масштабованість через CDN (Content Delivery Network) і підтримку адаптивного бітрейту (ABR), однак є неприйнятними для будь-якого інтерактивного застосунку. Розширення *LL-HLS* і *LL-DASH* скорочують затримку до 1-3 с через HTTP/2 chunked transfer та мікросегментацію, але і ці значення залишаються на порядок більшими за вимоги дистанційної підтримки.

Протокол RTMP гарантує менші затримки, проте поступово втрачає актуальність - розроблений компанією Adobe протокол поверх TCP із затримкою 1-5 с, що тривалий час використовувався для інгесту відео у трансляційні платформи. Нині поступово витісняється SRT і WebRTC через закритість специфікації та залежність від Flash-плеєра.

Стек RTSP/RTP широко використовується у системах відеоспостереження - це пара протоколів, де RTSP виступає протоколом керування («VCR-команди»: play, pause, record), а RTP - транспортом медіаданих поверх UDP. Латентність - порядку 2 с при достатній пропускній здатності мережі. Широко застосовується в IP-камерах, системах відеоспостереження та промисловій телематиці. Основне обмеження у браузерному контексті - відсутність нативної підтримки у браузерах.

Протокол SRT є сучасним рішенням для передачі відео через нестабільні мережі - відкритий протокол компанії Haivision поверх UDP із вбудованою корекцією помилок (FEC) і адаптивною буферизацією, розроблений для ненадійних WAN-каналів. Забезпечує затримку 0.5-2 с при збереженні стійкості на каналах із втратами до 25%. Підходить для «першої милі» (інгесту) у хмарний сервер, але не підтримується браузерами нативно.

Технологія WebRTC виступає єдиним з розглянутих рішень, що підтримує субсекундну затримку - єдиний з розглянутих протоколів, що нативно підтримується браузерами, забезпечує end-to-end шифрування (DTLS/SRTP) і досягає затримки менше 500 мс і навіть 100-300 мс у сприятливих мережевих умовах. Обмеженням є відносно низька масштабованість при P2P-топології та складність NAT-traversal. Порівняльна характеристика основних протоколів наведена у табл. 1.5 [31, 32].

Таблиця 1.5 - Характеристика відеостримінгових протоколів

Протокол	Транспорт	Типова затримка	ABR	Браузер (нативно)	Застосування
HLS / MPEG-DASH	TCP (HTTP)	6-30 с	Так	Так	VOD, лінійне ТБ

Протокол	Транспорт	Типова затримка	ABR	Браузер (нативно)	Застосування
LL-HLS / LL-DASH	TCP (HTTP/2)	1-3 с	Так	Так	Live-стримінг
RTMP	TCP	1-5 с	Ні	Ні	Інгест у студії
SRT	UDP	0.5-2 с	Ні	Ні	WAN-транспорт
RTSP/RTP	UDP	~2 с	Ні	Ні	IP-камери, CCTV
WebRTC	UDP (RTP)	< 0.5 с	Обмежено	Так	Відеозв'язок, AR

Оскільки WebRTC є протоколом вибору для інтерактивних систем із низькою затримкою, а різні архітектурні топології його застосування мають принципово різні характеристики. Аналіз топологічних моделей WebRTC-архітектур виділяє три основні конфігурації:

Модель P2P Mesh передбачає пряме з'єднання між учасниками сесії. Кожен учасник встановлює окреме RTCPeerConnection до кожного іншого. Медіасервер відсутній; необхідний лише сигнальний сервер для обміну SDP та ICE-кандидатами. Мінімальна затримка, мінімальне серверне навантаження і максимальна простота реалізації роблять цю модель ідеальною для сесій двох учасників. Суттєве обмеження - квадратичне зростання вихідного навантаження на клієнт при збільшенні кількості учасників: кожен з N учасників повинен надіслати $N-1$ копій свого потоку, що при $N > 4$ швидко вичерпує ресурси мобільних пристроїв.

Архітектура SFU базується на центральному вузлі, що перенаправляє потоки (але не обробляє) іншим учасникам. Кожен клієнт надсилає лише один висхідний потік, сервер масштабує розсилку. SFU підтримує simulcast (одночасна передача кількох версій потоку різної якості) і дозволяє клієнту вибирати якість отримуваного потоку залежно від умов мережі. Це на сьогодні домінантна архітектура у відеоконференційних платформах - Zoom, Google Meet, Discord використовують SFU або його гібридні варіанти [33-35]. Один SFU-сервер здатний

обслуговувати 500-1000 одночасних учасників; кластеризація розширює ліміт до десятків тисяч.

Архітектура MCU передбачає змішування медіапотоків на стороні сервера в один вихідний. Кожен клієнт надсилає один потік і отримує один змішаний потік незалежно від кількості учасників, що мінімізує вимоги до клієнтського завантаження і спрощує відображення. Ціна цього - надзвичайно висока обчислювальна вартість на стороні сервера: декодування, компонування та повторне кодування всіх потоків у реальному часі вимагають значних CPU/GPU-ресурсів. MCU залишається актуальним для мобільних або слабких клієнтів, а також для систем запису змішаного відео.

Визначальне місце у побудові систем реального часу займає relay-архітектура. Поряд із трьома класичними топологіями WebRTC у практиці систем реального часу виникає потреба у *relay-архітектурі* - проміжному вузлі, що забезпечує перетворення або ретрансляцію потоку між гетерогенними транспортами. Relay принципово відрізняється від SFU: він не обов'язково є WebRTC-учасником і може виступати «мостом» між різними протоколами та мережевими середовищами. Типові сценарії застосування relay: перетворення RTMP → WebRTC, TCP-WebSocket → UDP-RTP, або забезпечення зв'язку між пристроями у різних NAT-доменах.

У контексті систем дистанційної підтримки relay-компонент набуває особливого значення при необхідності підтримки гетерогенних кінцевих пристроїв. Якщо джерело відео (наприклад, автономна гарнітура або промисловий пристрій) не підтримує WebRTC нативно, relay-вузол виступає проксі: приймає потік від пристрою за власним протоколом, а до оператора у браузері ретранслює вже через WebRTC. Це дозволяє оптимізувати транспорт на кожному відрізку маршруту незалежно від можливостей кінцевих пристроїв.

Вибір кодеків та оцінка їх впливу на затримку є критичним фактором для проєктування, оскільки від кодека залежать обчислювальні витрати на кодування/декодування, якість при заданому бітрейті та вбудована затримка кодера.

H.264 (AVC) залишається де-факто стандартом для WebRTC-застосунків завдяки апаратній підтримці майже у всіх мобільних SoC та браузерах. Codec pipeline latency при потоковому кодуванні в режимі «zero-latency» (без B-кадрів, розмір GOP = 1) становить менше 33 мс на кадр при 30 FPS. Апаратні кодувальники Qualcomm Snapdragon XR2 Gen 2 (процесор Meta Quest 3S) підтримують H.264 із апаратним прискоренням, що забезпечує кодування 1080p при навантаженні менше 15% CPU.

H.265 (HEVC) забезпечує приблизно вдвічі вищу ефективність стиснення при однаковій якості порівняно з H.264, проте відрізняється вищою обчислювальною вартістю і менш повсюдною апаратною підтримкою у браузерах. Safari на Apple Silicon декодує HEVC апаратно, однак Chrome і Firefox не підтримують HEVC нативно у WebRTC.

VP8/VP9 - відкриті кодеки Google, що гарантовано підтримуються усіма WebRTC-браузерами. VP9 при однаковому бітрейті забезпечує якість, що наближається до H.265, але потребує більш потужного кодувальника. У сценаріях із обмеженими ресурсами мобільних пристроїв H.264 із апаратним кодуванням зазвичай переважає VP9 за швидкодією.

JPEG-over-UDP - нестандартний «кодек» у термінах відеостримінгу, що полягає у передачі послідовності окремих JPEG-кадрів через UDP. Кожен кадр стискається незалежно (внутрішньокадрове кодування, аналог I-frame-only у H.264), що усуває міжкадрові залежності і мінімізує затримку декодування до часу обробки одного кадру. Відсутність між-кадрового передбачення (motion estimation) є суттєвим недоліком з точки зору ефективності стиснення: JPEG-потік за однакового бітрейту поступається H.264 у якості приблизно у 3-5 разів. Проте відсутність GOP-залежностей, відсутність необхідності в decoder state і тривіальна реалізація на будь-якій платформі роблять цей підхід привабливим для систем, де спрощення інтеграції і детермінованість затримки важливіші за ефективність стиснення.

Для систем із змінними мережевими умовами критично важливим є механізм *адаптивного бітрейту* (ABR), що динамічно коригує параметри кодування

відповідно до поточної пропускної здатності каналу. У WebRTC ABR реалізується через механізм REMB (Receiver Estimated Maximum Bitrate) або TWCC (Transport-wide Congestion Control, RFC 8888) - зворотні сигнали від одержувача до відправника, що несуть оцінку доступної пропускної здатності. На основі цих сигналів кодувальник знижує або підвищує роздільну здатність, частоту кадрів або квантизацію.

Управління *відтворювальним буфером* (playout buffer) є ключовим компромісом між затримкою і плавністю відтворення. Великий буфер усуває джиттер і забезпечує плавне відео, але збільшує затримку. Мінімальний буфер дає мінімальну затримку, але веде до «заморожень» при пакетних втратах. Дослідження у галузі low-latency стримінгу показують, що для інтерактивних систем оптимальний розмір буфера становить 50-200 мс, тоді як для систем односторонньої трансляції - 1-5 с [36-38].

Розгляд хмарних, граничних та гібридних топологій розгортання виявляє такі особливості:

Хмарна модель (Cloud) розміщує весь серверний стек у публічній хмарі (AWS, GCP, Hetzner, тощо). Перевагами є глобальна доступність, еластичне масштабування і простота операційного управління; недоліком - добавлена мережева затримка між пристроями і хмарним вузлом, що залежить від географічної відстані та якості ISP.

Гранична модель (Edge Computing) переміщує обробку потоку до вузла, максимально наближеного до джерела або споживача. Це може бути корпоративний сервер у локальній мережі, міський точок присутності (PoP) або навіть смартфон-посередник. Затримка на ділянці пристрій → edge-вузол мінімізується, проте управління розподіленою граничною інфраструктурою значно складніше.

Гібридна модель поєднує обидва підходи: локальний (LAN) шлях використовується для з'єднань у межах однієї мережі із мінімальною затримкою, хмарний шлях активується як резервний при необхідності роботи через інтернет. Ця модель є архітектурно оптимальною для систем, що повинні функціонувати і в корпоративних інтранетах з ізольованим доступом, і у відкритому інтернеті,

оскільки вона використовує переваги обох середовищ без прив'язки до жодного з них. Переключення між шляхами може бути або ручним (конфігураційний параметр при розгортанні), або автоматичним - на основі успішності ICE-узгодження чи вимірювання RTT до хмарного вузла.

Процеси управління станом відеосесії та синхронізація медіаданих є необхідними для коректної роботи системи. Окремим аспектом клієнт-серверної архітектури для стримінгу є управління станом сесії - сукупністю метаданих про поточне з'єднання, кодувальні параметри, якість каналу та стани учасників. На відміну від stateless HTTP-запитів, WebRTC і WebSocket-з'єднання є stateful: розрив з'єднання руйнує стан пари RTCPeerConnection, і відновлення вимагає повторного проходження ICE-процедури. Це визначає необхідність реалізації механізмів *reconnect* з exponential backoff (послідовні спроби відновлення з подвоєнням інтервалу очікування), щоб запобігти «reconnect storm» - ситуації, коли безліч клієнтів одночасно атакують сервер після відновлення каналу.

Для систем із двонаправленим потоком (відео від виконавця до оператора і команди у зворотному напрямку) важливим є принцип незалежності каналів: вихід із ладу каналу керуючих команд не повинен призводити до зупинки відеопотоку, і навпаки. Ця незалежність досягається через використання окремих транспортів або окремих WebRTC DataChannels для медіа- та управляючих даних, з індивідуальними heartbeat-механізмами для кожного з них.

Забезпечення безпеки транспортного рівня у відеосистемах є першочерговим завданням при розробці. Безпека медіапотоку охоплює два аспекти: автентифікацію учасників сесії та конфіденційність переданих даних. WebRTC вирішує задачу конфіденційності на рівні протоколу: DTLS-рукостискання є обов'язковим відповідно до специфікації, і весь медіапотік шифрується SRTP (AES-128 або AES-256), а канали даних - SCTP/DTLS. Сигнальний канал WebSocket також має бути захищений TLS (протокол WSS), оскільки саме через нього передаються SDP-описи та ICE-кандидати - маніпуляція якими дозволила б провести атаку типу man-in-the-middle.

Для нестандартних протоколів, зокрема UDP-стримінгу поза WebRTC, шифрування не є вбудованим і повинно реалізовуватися на прикладному рівні або делегуватися мережевому рівню (VPN, TLS-тунель). Відсутність вбудованого шифрування в UDP є прийнятною у закритих LAN-середовищах, але неприйнятною для передачі через публічну мережу, де пасивне прослуховування не потребує активного втручання у маршрутизацію.

1.6. Безпека каналів зв'язку в телекомунікаційних системах реального часу

Розробка моделі загроз для систем реального часу базується на аналізі специфічних ризиків. Телекомунікаційні системи реального часу, що передають аудіовізуальний контент і керуючі команди через публічні або напівпублічні мережі, є об'єктом специфічного класу загроз, відмінного від загроз для традиційних HTTP-застосунків. Модель загроз (threat model) для таких систем охоплює три фундаментальні осі: конфіденційність (confidentiality) - запобігання несанкціонованому читанню медіаданих і метаданих сесії, цілісність (integrity) - запобігання модифікації потоку даних у транзиті, та доступність (availability) - запобігання переривання сесії через атаки відмови в обслуговуванні.

Специфіка систем реального часу полягає у тому, що традиційні захисні механізми з'являються в умовах жорстких часових обмежень: будь-яка криптографічна операція, що додає понад 10-20 мс затримки до кожного медіакадру, є практично неприйнятною. Це визначає особливі вимоги до вибору алгоритмів і архітектури захисту: перевагу мають симетричні шифри з апаратним прискоренням (AES-NI, ARM Cryptography Extensions) над обчислювально важкими асиметричними операціями, які виносяться у фазу ініціалізації з'єднання. Крім того, оскільки медіапотік передається переважно по UDP без гарантії

доставки, механізми захисту мають бути стійкими до втрат пакетів - тобто безпека кожного пакету не повинна залежати від успішної доставки попереднього.

Аналіз основних класів атак на системи MR-підтримки виявляє наступні загрози:

Пасивне прослуховування (eavesdropping) є найбільш поширеною атакою на незашифровані медіапотоки. Зловмисник, що знаходиться у тому самому мережевому сегменті (Wi-Fi, корпоративна LAN) або може перехопити трафік на маршрутизаторі, здатний записати весь потік UDP-пакетів і відновити з них відеозображення чи аудіо без будь-якого активного втручання. Пасивна атака не залишає слідів у журналах і не впливає на якість з'єднання, що ускладнює її виявлення.

Атака «людина посередині» (Man-in-the-Middle, MITM) є активнішим сценарієм: зловмисник не лише перехоплює, але й може модифікувати потік даних у транзиті або підмінити учасника сесії. У контексті WebRTC типовим вектором є атака на сигнальний канал: якщо обмін SDP-описами відбувається незашифрованим WebSocket (ws://), зловмисник може підмінити ICE-кандидати або модифікувати SDP-параметри кодеку, спрямувавши медіапотік через свій вузол. Захистом є обов'язкове використання захищеного WebSocket (wss://) і перевірка цілісності SDP-дескрипторів.

Захоплення сесії (session hijacking) у контексті WebSocket-з'єднань реалізується через атаку Cross-Site WebSocket Hijacking (CSWSH) - аналог CSRF для WebSocket [44, 46]. Якщо сервер автентифікує WebSocket-з'єднання виключно на основі HTTP-кукі без перевірки CSRF-токена або заголовку Origin, зловмисник може змусити браузер жертви встановити WebSocket-з'єднання до цільового сервера від імені авторизованого користувача. Захист полягає у перевірці заголовка Origin при рукостисканні, запровадженні токенів у URL з'єднання або використанні субпротокольної автентифікації.

Відмова в обслуговуванні (Denial of Service, DoS) для WebRTC-серверів може реалізовуватися через надсилання спеціально сформованих UDP-пакетів до відкритих медіапортів. Дослідники Enable Security (2024) [43, 45] описали DoS-

вразливість, яка використовує некоректні cipher suites у DTLS-запиті: оброблення такого пакету викликає DTLS-рівневу помилку, що блокує встановлення SRTP master keys і унеможливлює медіасесію - без жодної автентифікації зловмисника. Оскільки UDP-порти медіасерверів доступні публічно для ICE-connectivity checks, атакуючий може ефективно просканувати їх і вивести сервер з ладу відносно невеликим потоком пакетів.

Витік IP-адреси через WebRTC (WebRTC IP leakage) - специфічна вразливість браузерного WebRTC, за якої ICE-процес розкриває локальні та публічні IP-адреси клієнта навіть при використанні VPN. Під час збирання ICE-кандидатів браузер передає всі мережеві інтерфейси STUN-серверу, включаючи внутрішні адреси, що ідентифікують мережеве оточення користувача. Сучасні браузери частково пом'якшили цю проблему через «mDNS-обфускацію» ICE-кандидатів (заміна реальних IP на mDNS-імена), однак повністю усунути витік без відключення WebRTC на рівні браузерних налаштувань неможливо.

Replay-атаки полягають у записі легітимних медіапакетів і їх повторній передачі через деякий час для імітації учасника або порушення стану сесії. Без захисту від replay зловмисник може, наприклад, повторно відтворити застарілі анотаційні команди, що змінить стан відображення у виконавця.

Використання протоколу TLS забезпечує надійний захист сигнального каналу. TLS (Transport Layer Security) є фундаментальним протоколом захисту транспортного рівня для TCP-з'єднань, включаючи HTTP (HTTPS) і WebSocket (WSS). Поточною актуальною версією є TLS 1.3, стандартизована у RFC 8446 (2018), яка скорочує кількість round-trips при рукостисканні до одного (1-RTT) або нуля (0-RTT при відновленні сесії) порівняно з двома у TLS 1.2. TLS 1.3 виключає підтримку відомо небезпечних алгоритмів: RSA key exchange, RC4, MD5, SHA-1, а також режими CBC для симетричного шифрування, що були вразливі до атак BEAST і POODLE.

Для WebSocket-з'єднань перехід від ws:// до wss:// є не опцією, а мінімальним обов'язковим заходом безпеки: незашифрований WebSocket передає всі дані - включаючи SDP-описи, ICE-кандидати і керуючі команди - у відкритому тексті. При

цьому WSS забезпечує лише безпеку транспорту, але не замінює автентифікацію додатку: сервер повинен додатково перевіряти автентичність клієнта через токени (JWT, API keys) або механізм OAuth.

Механізм DTLS виступає як адаптація TLS для роботи з датаграмним транспортом. DTLS (Datagram Transport Layer Security) є адаптацією TLS для UDP-транспорту, стандартизованою у RFC 6347 (DTLS 1.2) і RFC 9147 (DTLS 1.3). Ключова відмінність від TLS полягає у необхідності обробки пакетних втрат і переупорядкування: DTLS вводить явні sequence numbers у записи та механізм повторної передачі рукостискального трафіку з таймаутами, що дозволяє завершити handshake навіть при значному відсотку втрат.

DTLS у WebRTC використовується виключно у фазі ініціалізації з'єднання: через DTLS-рукостискання узгоджуються криптографічні ключі і параметри сесії, після чого для медіатрафіку задіюється SRTP (де ключовий матеріал отримано саме з DTLS-рукостискання за механізмом DTLS-SRTP, RFC 5764). Таким чином, DTLS не шифрує медіадані безпосередньо - він лише забезпечує автентифікований і захищений обмін ключами перед початком передачі. Специфікація WebRTC (RFC 8827) [39-41] вимагає, щоб DTLS-рукостискання підтверджувало фінгерпринт сертифікату, зазначений у SDP-описі, - це захищає від атак MITM на рівні медіатранспорту навіть у разі, якщо сигнальний канал скомпрометований.

Протокол SRTP використовується для наскрізного шифрування медіапотоку. SRTP (Secure Real-time Transport Protocol), визначений у RFC 3711, є розширенням RTP з підтримкою шифрування, автентифікації повідомлень і захисту від replay-атак для медіапотоків. Шифрування здійснюється за алгоритмом AES у режимах CTR або GCM з ключами довжиною 128 або 256 біт; режим GCM (Galois/Counter Mode, AES-GCM) є автентифікованим шифруванням (AEAD), що одночасно забезпечує і конфіденційність, і цілісність кожного пакету.

Критично важливою властивістю SRTP є replay protection: кожен пакет містить порядковий номер (sequence number), і одержувач веде ковзне вікно прийнятих номерів. Пакети з вже прийнятими номерами відкидаються, що унеможливлює replay-атаки. Обчислювальна вартість SRTP при апаратній

підтримці AES-NI є мінімальною - менше 1% CPU на сучасних мобільних процесорах - що підтверджує придатність протоколу для систем з жорсткими вимогами до затримки.

Слід, однак, відзначити, що SRTP у базовій специфікації не шифрує заголовки RTP-пакетів - вони передаються у відкритому вигляді. Заголовки містять SSRC (ідентифікатор джерела), тимчасові мітки і порядкові номери, що дозволяє аналізувати трафік і робити висновки про активність учасників без доступу до медіавмісту. Цю проблему вирішує стандарт «Cryptex» (RFC 9335) [42], що забезпечує повне шифрування заголовків розширень і CSRC-полів SRTP.

Забезпечення безпеки нестандартних UDP-каналів потребує впровадження додаткових перевірок. Для UDP-транспорту поза WebRTC - зокрема для власних протоколів стримінгу - вбудовані механізми безпеки відсутні. Захист у таких випадках може досягатися кількома підходами: мережевим рівнем (VPN-тунелювання, IPsec), прикладним шифруванням (шифрування корисного навантаження пакету засобами AES-GCM до передачі), або архітектурним обмеженням (обмеження доступу до UDP-портів лише довіреними мережевими адресами через файрволл).

Прикладне шифрування UDP-пакетів має ряд нюансів: на відміну від DTLS, відсутній механізм узгодження ключів, тому ключовий матеріал повинен бути попередньо розподілений або переданий через захищений канал (наприклад, через TLS-з'єднання під час реєстрації пристрою). Крім того, розмір заголовка шифрування (IV, тег автентифікації GCM) збільшує розмір пакету на 28-44 байти, що при фрагментованій передачі великих фреймів помітно підвищує накладні витрати.

Важливим аспектом є файрвольна фільтрація UDP-портів: оскільки UDP-сокети не мають вбудованого механізму автентифікації з'єднання (на відміну від TCP SYN-рукостискання), будь-хто, хто знає відкритий UDP-порт, може надсилати туди дані. Це відкриває вектор для атак підробки адреси відправника (UDP spoofing) і DoS-ампліфікації, якщо сервіс повертає відповіді більші за запити.

Процеси автентифікації учасників та управління сесіями дозволяють обмежити доступ до системи. Криптографічний захист транспорту є необхідною, але недостатньою умовою безпеки системи. Рівнозначно важливою є автентифікація учасників - підтвердження того, що до сесії підключаються саме ті пристрої і користувачі, які мають на це право. У WebRTC-архітектурах автентифікація реалізується на рівні сигнального сервера - до того, як відбувається обмін SDP. Типові механізми: Bearer-токени (JWT) у WebSocket upgrade-запиті, попередньо поширені секрети (TURN credentials) або сертифікати клієнта у DTLS-рукостисканні.

Для TURN-серверів специфікація передбачає механізм time-limited credentials (RFC 5389): TURN-ім'я користувача кодує час закінчення дії, а пароль є HMAC-SHA1 від цього імені і спільного секрету. Це означає, що навіть якщо TURN-credentials будуть перехоплені, вони стають непридатними після закінчення терміну дії (зазвичай 24 години). Сервери coturn і Twilio TURN реалізують цей механізм за замовчуванням.

Специфічною вразливістю є *витік IP-адреси через WebRTC*, що може розкрити топологію мережі. Системи дистанційної MR-підтримки за своєю архітектурою є ближчими до IoT-сценаріїв, ніж до класичних відеоконференційних платформ: один із кінцевих пристроїв (гарнітура) є вбудованим комп'ютером з обмеженими ресурсами, функціонуючим у нетипових мережевих умовах (Wi-Fi у промисловому середовищі, хот-спот смартфона), і часто взаємодіє з проміжним relay-вузлом (смартфон чи планшет). У таких гетерогенних топологіях забезпечення наскрізного шифрування (end-to-end encryption) ускладнюється: relay-вузол, що перетворює протоколи, за визначенням повинен мати доступ до транспортованих даних у формі «plaintext», що обриває суворий E2E-ланцюжок.

Дослідження у галузі безпеки хмарних IoT-архітектур [47, 48] рекомендують застосовувати multi-protocol layered security: окреме шифрування на кожному відрізку (гарнітура → relay, relay → хмара, хмара → браузер), де кожна ланка захищається найбільш відповідним для неї протоколом - UDP з прикладним

шифруванням, DTLS або TLS - із розумінням того, що relay-вузол є довіреним компонентом системи, а не зовнішньою стороною.

ІІ. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ ПІДТРИМКИ НА ОСНОВІ ЗМІШАНОЇ РЕАЛЬНОСТІ

2.1. Концепція системи V.I.R.A. та обґрунтування архітектурних рішень

За своїм призначенням та концепцією, V.I.R.A. (Virtual Interactive Remote Assistance) — це клієнт-серверна система дистанційної експертної допомоги, що забезпечує двосторонню просторову комунікацію між оператором-експертом, який перебуває на віддаленому робочому місці, та користувачем на місці події, оснащеним гарнітурою змішаної реальності Meta Quest 3S. Ключова ідея системи полягає в поєднанні відеопотоку від пристрою MR з можливістю накладання просторових анотацій у world-space координатах сцени, що дозволяє оператору вказувати на конкретні об'єкти реального середовища - на відміну від screen-space рішень, у яких анотації «прив'язані» до пікселів відеокадру, а не до фізичних точок простору. Система проєктувалась як self-hosted: усі компоненти розгортаються без залежності від зовнішніх хмарних платформ, що відповідає вимогам до ізольованих виробничих мереж та режиму роботи без постійного інтернет-з'єднання.

Щодо функціональних та нефункціональних вимог, у процесі проєктування V.I.R.A. було сформульовано такі ключові критерії до системи:

- *Передача відеопотоку* з гарнітури Quest 3S на браузерний інтерфейс оператора з наскрізною затримкою (glass-to-glass latency) не вище 250-400 мс, що відповідає порогу сприйняття тактильного зворотного зв'язку (підрозділ 1.3);
- *Двосторонні анотації*: оператор малює мітки (штрихи, текст, семантичні маркери M1-M7) у браузері, користувач бачить їх накладеними на реальне середовище у world-space на гарнітурі;
- *Двошляхова транспортна архітектура*: система повинна функціонувати як у локальній мережі (LAN-режим, без доступу до Інтернету), так і через хмарний ретранслятор (Cloud-режим, NAT-traversal через coturn/TURN);

- *Мінімальні вимоги до обладнання на місці*: гарнітура Quest 3S та Android-смартфон для relay-з'єднання у хмарному режимі; встановлення спеціалізованого ПЗ не потрібне;
- *Self-hosted розгортання*: сервер Node.js та TURN-сервер coturn розміщуються на будь-якій хмарній або локальній машині (у реалізації - Hetzner CAX11 ARM);
- *Безпека каналу*: шифрування TLS/WSS на WebSocket-каналі, DTLS-SRTP на WebRTC-каналі (підрозділ 1.6).

Архітектурно система V.I.R.A. реалізована у вигляді чотирьох взаємодіючих компонентів:

Компонент 1 - Unity-застосунок (Meta Quest 3S). Написаний на C# для платформи Android/Meta XR SDK. Реалізує захоплення passthrough-відеопотоку через OpenXR Passthrough Layer та OVRPassthroughLayer, кодування кадрів у JPEG у фоновому потоці через ImageConversion.EncodeArrayToJPG + ThreadPool.QueueUserWorkItem, фрагментацію пакетів (до 60 000 байт/UDP-пакет) та відправлення через UDP (порт 50501). Паралельно цей компонент приймає анотації від сервера через UDP (порт 50502) та відображає їх у world-space за допомогою OVRSpatialAnchor + LineRenderer.

Компонент 2 - Android Relay (Kotlin, Android 8.0+). Мостовий застосунок на телефоні або планшеті, що знаходиться в одній Wi-Fi-мережі з гарнітурою. Реалізує: UDP Discovery (broadcast 255.255.255.255:50500) для автоматичного виявлення гарнітури; збирання фрагментованих UDP-пакетів у цілісні JPEG-кадри через FrameAssembler; пересилання зібраних кадрів на хмарний сервер через WebSocket (OkHttp); зворотне пересилання JSON-анотацій від сервера до гарнітури через UDP (порт 50502). У хмарному режимі Android Relay є єдиним мостом між UDP-мережею гарнітури та WebSocket/WebRTC-інфраструктурою Інтернету.

Компонент 3 - Хмарний сервер (Node.js + TypeScript + coturn). Серверна частина складається з backend-застосунку (Express + ws-бібліотека) та TURN-сервера coturn. Backend приймає відеофрейми від Android Relay (або напряму від Quest у LAN-режимі) через WebSocket, розподіляє їх підключеним браузерним

клієнтам, обробляє JSON-анотації у зворотному напрямку, а також реалізує WebRTC-сигналінг (offer/answer/ICE через конверт webrtcsignal). Coturn забезпечує TURN-relay для NAT-traversal при встановленні peer-to-peer WebRTC-сесії між Android Relay та браузером.

Компонент 4 - Браузерний інтерфейс оператора (React + TypeScript + Vite). Single-page application, що відображає відеопотік на Canvas-елементі (VideoCanvas.tsx), надає інструменти малювання анотацій (штрихи, текст, семантичні маркери M1-M7 через MarkerPalette), підтримує функцію Freeze Frame (зупинка кадру для точного позиціонування анотацій), реалізує WebRTC-сесію (RTCPeerConnection) та WebSocket-підключення до сервера. Інтерфейс підтримує мультимовний режим (UA/EN/DE/PL/PT/JP) та темну/світлу тему.

Обґрунтування вибору саме чотирикомпонентної топології зумовлене обмеженнями існуючих комерційних рішень, детально проаналізованих у підрозділі 1.1. Як зазначено у висновках підрозділу 1.1, усі розглянуті системи (MS Dynamics 365 Remote Assist, TeamViewer Assist AR, PTC Vuforia Chalk, Scope AR WorkLink) мають спільні структурні недоліки: жорстка прив'язка до хмарних SaaS-платформ (Azure, Office 365) унеможливлює роботу в ізольованих мережах; відсутність підтримки video passthrough-гарнітур (зокрема Meta Quest 3S) обмежує клас підтримуваних пристроїв; screen-space анотації у TeamViewer Assist AR та PTC Vuforia Chalk втрачають просторову прив'язку при русі користувача; відсутній власний транспортний стек - залежність від WebRTC TURN-хмар стороннього постачальника (підрозділ 1.1).

V.I.R.A. усуває кожен з цих недоліків: Unity-застосунок побудований виключно на Meta XR SDK та OpenXR без хмарних залежностей; world-space анотації з OVRSpatialAnchor забезпечують просторову стабільність навіть при русі голови (підрозділи 1.4, 2.4); Android Relay дозволяє використовувати стандартний Android-пристрій як мережевий міст без модифікації гарнітури чи мережевої інфраструктури; self-hosted coturn і Node.js сервер повністю виключають залежність від SaaS-постачальника. Відсутність native WebRTC-підтримки у Unity для Meta Quest 3S (пакет com.unity.webrtc несумісний із standalone Android/Quest

збіркою) унеможливлює пряме P2P-з'єднання гарнітури з браузером, що й обумовлює необхідність проміжного Android Relay як UDP-WebSocket/WebRTC шлюзу.

Центральним архітектурним рішенням V.I.R.A. є двошляховий транспорт, тобто підтримка двох режимів транспорту в межах єдиної кодової бази, керованих змінною середовища VIRA_TRANSPORT_MODE:

- LAN-режим (localudpws): Quest 3S надсилає JPEG-фрейми напряму на Node.js-сервер через UDP (порт 50501); сервер ретранслює їх у браузер через WebSocket. Анотації повертаються UDP (порт 50502) напряму на гарнітуру. Android Relay у цьому режимі не задіяний. Затримка - мінімальна (≈ 5-20 мс LAN RTT), відсутня залежність від Інтернету;
- Cloud-режим (cloudwebrtc): Quest 3S надсилає UDP-пакети на Android Relay; Relay ретранслює їх через WebSocket wss://relay.vira.example/relay на хмарний сервер; браузер отримує відеопотік через WebRTC-медіадоріжку (після ICE/DTLS handshake з coturn як TURN-сервером). Цей режим забезпечує роботу через NAT та за відсутності прямого IP-з'єднання між гарнітурою та оператором.

Така дворівнева схема обрана усвідомлено: LAN-режим забезпечує максимальну продуктивність та ізольованість у промислових умовах (що критично, наприклад, у задачах, пов'язаних з роботою в зонах без доступу до Інтернету), тоді як Cloud-режим реалізує повноцінне дистанційне супроводження з будь-якої точки світу. Архітектурна уніфікація двох режимів в одному серверному застосунку мінімізує операційну складність розгортання, оскільки оператор перемикає режим зміною єдиної змінної середовища без перекомпіляції компонентів (детально - підрозділ 2.8).

Схема наведена в Додатку А фіксує повний маршрут даних для обох транспортних режимів і слугує базисом для детального розгляду кожного підкомпоненту в наступних підрозділах (2.2-2.8).

2.2. Захоплення та стиснення відеопотоку на гарнітурі Meta Quest 3S

З огляду на вихідні умови та обмеження платформи, варто зазначити, що Meta Quest 3S є standalone-пристроєм на базі Qualcomm Snapdragon XR2 Gen 2, відеопотік з фізичних камер якого не є безпосередньо доступним для Unity-застосунку через стандартний WebCamTexture-інтерфейс. Єдиним підтримуваним способом програмного доступу до зображень passthrough-камер є Passthrough Camera API у складі Meta XR SDK, що надає клас PassthroughCameraAccess з методами отримання поточного кадру, пози та внутрішніх параметрів камери. Цей API надає знімки у форматі YUV420 з роздільністю до 1280×960 пікселів при затримці захоплення 20-40 мс (підрозділ 1.2). Оскільки Unity-рушій використовує власний GPU-конвеєр рендерингу, а PassthroughCameraAccess передає зображення у вигляді GPU-текстури (а не у CPU-пам'яті), весь pipeline захоплення побудований навколо механізму асинхронного зчитування пікселів з GPU - AsyncGPUReadback.

Ключовим архітектурним рішенням AR-композитного конвеєра захоплення є те, що в мережу передається не «чисте» passthrough-зображення, а AR-композит - кадр із вже нанесеними world-space анотаціями. Це означає, що оператор у браузері бачить рівно те саме, що бачить користувач у гарнітурі, включно з усіма накладеними елементами. Реалізація цього механізму покладається на спеціальну камеру VIRACaptureCamera, яка динамічно створюється при ініціалізації компонента PassthroughSenderClean і синхронізується з поточною позою passthrough-камери через метод SyncCaptureCamera().

Метод SyncCaptureCamera() зчитує з PassthroughCameraAccess.Intrinsics фактичні оптичні параметри камери гарнітури: фокусні відстані f_x, f_y, головну точку (c_x, c_y) та роздільність сенсора (w, h). На їх основі будується явна матриця проєкції (2.1):

$$P = \begin{pmatrix} \frac{2f_x}{w} & 0 & 1 - \frac{2c_x}{w} & 0 \\ 0 & \frac{2f_y}{h} & 1 - \frac{2c_y}{h} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (2.1)$$

де n та f - відстані до ближньої та дальньої площин відсікання. Таке явне задання матриці проєкції замість стандартного camera.fieldOfView є обов'язковим для правильного суміщення AR-анотацій із passthrough-зображенням: без врахування реальних внутрішніх параметрів оптики виникає геометричне зміщення між накладеними об'єктами та відповідними фізичними точками сцени.

У рамках покадрового конвеєра захоплення кожен кадр обробляється корутиною CaptureAndSendAsync(), яка запускається з Update() з інтервалом не менше frameInterval = 1/framesPerSecond та лише якщо прапорець isProcessing == false (що запобігає накопиченню черги при зниженні FPS). Конвеєр складається з таких кроків:

1. Blit passthrough-текстури до RenderTexture resizeRT (1280×960, формат ARGB32) через Graphics.Blit(sourceRawImage.texture, resizeRT). На цьому кроці виконується масштабування до цільової роздільності та конвертація колірного простору силами GPU.

2. AR-композит (якщо compositeAnnotations == true): синхронізація VIRACaptureCamera з поточною позою та параметрами камери (SyncCaptureCamera()), встановлення captureCamera.targetTexture = resizeRT, виклик captureCamera.Render(). Після цього в resizeRT містяться як passthrough-фон, так і всі world-space анотації (штрихи, маркери), оскільки cullingMask камери захоплення включає шар UI-об'єктів (виключаючи системний шар UI).

3. Асинхронне зчитування з GPU через AsyncGPUReadback.Request(resizeRT, 0, TextureFormat.RGBA32). Корутина блокується на yield return null до завершення операції читання, що дозволяє уникнути зупинки головного потоку при передачі даних з VRAM у RAM. Підсумком є масив byte[] rawBytes у форматі RGBA32.

4. JPEG-кодування у фоновому потоці: масив rawBytes та параметри кадру передаються до ThreadPool через ThreadPool.QueueUserWorkItem(...), де виконується ImageConversion.EncodeArrayToJPG(rawBytes, GraphicsFormat.R8G8B8A8_UNorm, (uint)w, (uint)h, 0, quality) з якістю jpegQuality = 75. Результат (масив байт JPEG) поміщається до потокобезпечної черги ConcurrentQueue<byte[]> jpegQueue.

5. Відправлення з головного потоку: у методі Update() черга jpegQueue вичерпується, кожний готовий JPEG-кадр передається у SendFragmentedFrame() для UDP-фрагментації та відправлення (детально - підрозділ 2.3).

Розподіл роботи між GPU (Blit + Render + AsyncGPUReadback) та ThreadPool (JPEG-кодування) є результатом вирішення проблеми продуктивності, зафіксованої у журналі розробки: початкова реалізація викликала EncodeToJPG у головному потоці, що спричиняло блокування рендерингу на 40-80 мс і знижувало FPS до менш ніж 10. Після переходу до AsyncGPUReadback + ThreadPool вдалося стабілізувати реальний FPS на рівні 18-25 кадрів/с при цільовому значенні 30 FPS. Глибший аналіз показав, що на ARM Snapdragon XR2 Gen 2 операція AsyncGPUReadback займає 14-42 мс (обумовлено частотою GPU-таймстемпів 72 Гц), а JPEG-кодування у ThreadPool - 40-80 мс; таким чином, головний потік звільнений від будь-яких блокуючих операцій.

Параметри захоплення та їх обґрунтування наведені у табл. 2.1, де вказані ключові конфігураційні параметри підсистеми захоплення.

Таблиця 2.1 - Конфігураційні параметри підсистеми захоплення відеопотоку

Параметр	Значення	Обґрунтування
targetWidth × targetHeight	1280 × 960 пікс.	Відповідає нативній роздільності passthrough-камери Quest 3S (1280×960); збереження нативного співвідношення сторін 4:3 виключає геометричні спотворення при AR-суміщенні
jpegQuality	75	Емпіричний компроміс: якість достатня для впізнавання деталей сцени, розмір кадру ~12-18 КБ при 1280×960; підвищення до 90 збільшує розмір вдвічі без суттєвого приросту чіткості

Параметр	Значення	Обґрунтування
framesPerSecond	30	Цільове значення; фактичне ~18-25 FPS обумовлено часом AsyncGPUReadback; 30 FPS є мінімально прийнятним для відстеження руху руки (порогове значення ~100 мс затримки для тактильного відгуку, підрозділ 1.3)
MAX_UDP_PACKET_SIZE	60 000 байт	Менше максимального розміру UDP-дейтаграми (65 507 байт), але вище типового MTU Ethernet (1500 байт); фрагментація на мережевому рівні виконується IP-стеком ОС Android, а не Unity (детально - підрозділ 2.3)
RenderTextureFormat	ARGB32	Мінімальний формат, що підтримує альфа-канал для AR-композиту; RGBA32 при AsyncGPUReadback - єдиний стабільний формат на GLES3 (Android/Quest)

Для обґрунтування вибору JPEG як формату стиснення варто згадати підрозділ 1.5, де розглянуто кодеки, що застосовуються у системах стримінгу низької затримки. Зокрема, зазначено, що JPEG-over-UDP є I-frame-only форматом без залежності від GOP-структури та decoder state, що відрізняє його від H.264, де будь-яка втрата P- або B-кадру вимагає повного перезапиту декодерного стану. Також зафіксовано, що H.264 забезпечує у 3-5 разів меншу бітову швидкість за рахунок міжкадрового передбачення (motion estimation), але вимагає апаратного кодека або значних ресурсів CPU для програмної реалізації (підрозділ 1.5).

Для застосунку на Meta Quest 3S вибір JPEG є єдиним практично обґрунтованим рішенням з таких причин:

- Першим фактором є відсутність доступу до апаратного H.264-кодека з Unity. Пакет com.unity.webrtc, що міг би надати доступ до VideoStreamTrack з апаратним H.264-кодуванням через Qualcomm codec pipeline, несумісний зі standalone Android/Quest збіркою у версіях Unity, що використовуються в проєкті (Unity 6, Meta XR SDK 85.0.0). Спроба інтеграції com.unity.webrtc зафіксована як заблокована у ROADMAP як «post-prod out-of-scope». Програмне H.264-кодування

силами ffmpeg або аналогів на CPU Snapdragon XR2 Gen 2 при роздільності 1280×960 займає понад 100 мс на кадр, що є неприйнятним для реального часу.

- Другим фактором є відсутність decoder state на приймачі. Оскільки UDP не гарантує доставку та порядок пакетів, частина кадрів неминуче губиться. При H.264 втрата I-кадру або P-кадру у GOP-ланцюжку призводить до артефактів «заморозки» або «розпаду» зображення до наступного I-кадру. JPEG є самодостатнім у межах одного кадру: кожен JPEG-кадр декодується незалежно, а втрата окремого пакету призводить лише до пропуску одного кадру (або відображення останнього успішно отриманого) без накопичувальних артефактів (підрозділ 1.5).
- Третім фактором є простота реалізації в межах Unity без нативних плагінів. Алгоритм ImageConversion.EncodeArrayToJPG є частиною стандартного Unity API, не потребує нативних бібліотек і стабільно працює у середовищі ThreadPool на ARM без ризику IL2CPP-краш-ів, які виникали при спробах використати Texture2D.EncodeToJPG з фонових потоків.

Таким чином, JPEG-over-UDP для даної системи є не компромісним, а архітектурно мотивованим вибором, що враховує реальні обмеження платформи, відсутність стабільного WebRTC-відеокодека для Unity/Quest та вимогу до стійкості потоку при втраті UDP-пакетів.

2.3. Протокол фрагментованої UDP-передачі відеопотоку

Застосування власного прикладного протоколу обґрунтовується тим, що, як зазначено у підрозділі 1.3, максимальний розмір UDP-дейтаграми становить 65 507 байт (обмеження IPv4 з урахуванням заголовків IP та UDP), тоді як MTU стандартної мережі Ethernet дорівнює 1 500 байт, із яких 1 472 байт доступні для корисного навантаження UDP-пакету. Кадр JPEG розміром 40-100 КБ при роздільності 1280×960 і якості 75 не вкладається ні в один UDP-пакет без

фрагментації на мережевому рівні. IP-фрагментація, однак, є небажаним механізмом для мультимедійних потоків у реальному часі: втрата будь-якого IP-фрагменту призводить до відкидання всієї IP-датаграми на приймачі без можливості часткового відновлення, а повторна збірка виконується лише на кінцевому вузлі, що унеможливлює управління буфером на рівні застосунку (підрозділ 1.3). З цих причин у V.I.R.A. реалізовано власний прикладний протокол фрагментації, у якому розбиття та збирання кадру відбувається в коді застосунку, а кожен UDP-пакет є самостійною і повноцінною одиницею доставки.

Щодо формату заголовку пакету, кожен UDP-пакет відеопотоку V.I.R.A. складається з три-байтного заголовку та корисного навантаження. Структура заголовку наведена у таблиці 2.2.

Таблиця 2.2 - Структура заголовку фрагментованого UDP-пакета

Байт(и)	Поле	Тип	Діапазон	Призначення
0	frameId	uint8	0-255, циклічно	Ідентифікатор кадру; монотонно зростає, обнуляється після 255
1	chunkIndex	uint8	0-254, 0-based	Порядковий номер фрагмента в межах кадру
2	totalChunks	uint8	1-255	Загальна кількість фрагментів даного кадру
3..N	Payload	byte[]	max 60 000 байт	Частина JPEG-даних відповідного фрагмента

Загальний розмір пакету становить не більше 60 003 байт (3 байти заголовку + 60 000 байт корисного навантаження), що свідомо менше за максимальний допустимий розмір UDP-датаграми (65 507 байт). Обрана межа MAX_UDP_PACKET_SIZE = 60 000 байт залишає резерв ~5 507 байт відносно максимуму UDP і гарантує, що IP-фрагментація на мережевому рівні не відбувається за умови підтримки jumbo frames у локальній Wi-Fi-мережі; у звичайних умовах Wi-Fi (де MTU може бути нижчим) ОС Android самостійно фрагментує UDP-датаграму на IP-рівні з прозорою збіркою на приймачі, не порушуючи логіки прикладного протоколу.

Фрагментація реалізована у методі SendFragmentedFrame класу PassthroughSenderClean, що наведено у лістингу 2.1.

Лістинг 2.1 - Алгоритм фрагментації UDP-пакетів на стороні клієнта

```
private void SendFragmentedFrame(byte[] data) {
   int totalLength = data.Length;
   int chunkCount = (int)Math.Ceiling((double)totalLength / MAX_UDP_PACKET_SIZE);
   frameId++;                          // uint8, циклічний лічильник
   for (int i = 0; i < chunkCount; i++) {
      int offset = i * MAX_UDP_PACKET_SIZE;
      int length = Math.Min(MAX_UDP_PACKET_SIZE, totalLength - offset);
      byte[] header = new byte[3];
      header[0] = frameId;              // ID поточного кадру
      header[1] = (byte)i;              // chunkIndex (0-based)
      header[2] = (byte)chunkCount;     // totalChunks
      byte[] packet = new byte[3 + length];
      Buffer.BlockCopy(header, 0, packet, 0, 3);
      Buffer.BlockCopy(data, offset, packet, 3, length);
      videoSenderClient.Send(packet, packet.Length);
   }
}
```

Поле frameId є полем типу byte у C# і природно переповнюється після 255, знову починаючи з 0. При цільовій частоті 30 FPS повний цикл переповнення frameId становить приблизно 8,5 секунди - достатній інтервал, щоб на приймачі не виникло колізій між кадром з frameId = 0 (старий, незавершений) та frameId = 0 (новий): застаріли буфери видаляються раніше, ніж відбувається повторне використання ідентифікатора (детально - нижче).

Типовий JPEG-кадр при jpegQuality = 75 і роздільності 1280×960 займає 40-100 КБ, тобто потребує лише 1-2 UDP-пакетів. Це є важливою властивістю протоколу: оскільки переважна більшість кадрів передається одним пакетом, реальний відсоток ситуацій, де фрагментація відбувається на прикладному рівні, є низьким, і алгоритм збірки виконується переважно для повноти реалізації (на випадок кадрів з більшою деталізацією або вищою якістю JPEG). Загальний час відправлення фрагментованого кадру у SendFragmentedFrame за виміряними значеннями становить ~25 мс.

На стороні Android Relay клас FrameAssembler реалізує повну логіку збирання кадрів із вхідних UDP-пакетів (лістинг 2.2).

Лістинг 2.2 - Структури даних для збирання фрагментованих відеокадрів

```
private data class FrameBuffer(
    val totalChunks: Int,
    val chunks: MutableMap<Int, ByteArray> = mutableMapOf(),
    var receivedChunks: Int = 0,
    val timestamp: Long = System.currentTimeMillis()
)
private val frames = ConcurrentHashMap<Int, FrameBuffer>()  // ключ - frameId
```

Алгоритм обробки кожного вхідного UDP-пакету:

1. Розбирання заголовку: витягуються frameId, chunkIndex, totalChunks, payload.
2. Якщо запису з ключем frameId у frames немає - створюється новий FrameBuffer(totalChunks).
3. Payload зберігається у chunks[chunkIndex]; лічильник receivedChunks збільшується.
4. Якщо receivedChunks == totalChunks:

- Фрагменти впорядковуються за chunkIndex (0, 1, 2, …);
- Конкатенуються у єдиний масив байт (цілісний JPEG);

- Масив надсилається як бінарний WebSocket-фрейм на хмарний сервер (CloudWebSocket.sendBinary);
- FrameBuffer видаляється з frames.

Клас VideoReceiver приймає UDP-пакети на порту 50501 з буфером прийому recvBufferSize = 8 МБ у IO-потоці (Dispatchers.IO). Вибір розміру буфера 8 МБ обґрунтований виміряними характеристиками трафіку: при 25 FPS і середньому розмірі кадру ~60 КБ миттєва швидкість складає ~1,5 МБ/с, а буфер 8 МБ поглинає пікові сплески (~5 кадрів) без переповнення сокета і втрати пакетів на рівні ОС.

Для обробки втрат та застарілих буферів, оскільки UDP не гарантує доставку пакетів (підрозділ 1.3), у FrameAssembler реалізовано механізм автоматичного прибирання незавершених буферів:

- Stale timeout: кожен FrameBuffer зберігає мітку часу timestamp - момент отримання першого фрагмента кадру. Фоновий cleanup-цикл з інтервалом cleanupIntervalMs = 1 000 мс перевіряє всі активні буфери: якщо поточний час перевищує timestamp на staleTimeoutMs = 500 мс, буфер визнається застарілим і видаляється.
- Wrap-around безпека: часовий поріг 500 мс гарантує, що при частоті 30 FPS новий кадр з тим самим frameId (після обнулення через 255 кадрів, тобто через ~8,5 с) ніколи не потрапить у старий буфер - застарілий буфер буде прибраний щонайменше через 1 с після отримання першого фрагмента.
- Немає повторних запитів (no retransmit): якщо кадр не зібраний до stale timeout, він просто відкидається. Це є осмисленим рішенням: у потоці реального часу застарілий кадр не має цінності для відображення, а запит на повторну передачу вніс би RTT-затримку і порушив би природний хід відеопотоку. Такий підхід узгоджується з властивостями UDP, описаними у підрозділі 1.3, де зазначено, що відсутність механізму retransmission і head-of-line blocking є перевагою UDP для мультимедіа, а не недоліком.

Ця стратегія обробки втрат додатково підкріплюється тим, що JPEG є I-frame-only форматом: кожен кадр декодується незалежно (підрозділ 2.2), тому пропуск одного або декількох кадрів через stale-cleanup призводить лише до локального

«підстрибування» відображення (frame skip), але не до накопичувальних артефактів, характерних для GOP-кодеків. Зведені параметри розробленого протоколу фрагментації наведено у табл. 2.3.

Таблиця 2.3 - Зведені параметри протоколу фрагментації

Параметр	Значення	Обґрунтування
MAX_UDP_PACKET_SIZE	60 000 байт	Менше максимуму UDP (65 507), більше MTU Ethernet (1 472); баланс між кількістю пакетів на кадр та ризиком IP-фрагментації
Розмір заголовку	3 байти	Мінімально достатній для ідентифікації кадру та фрагмента; накладні витрати < 0,005% від розміру payload
frameId	uint8 (0-255)	Wrap-around ~8,5 с при 30 FPS; достатньо для однозначної ідентифікації до завершення stale cleanup
Stale timeout	500 мс	Більше одного міжкадрового інтервалу (33 мс при 30 FPS), але менше за будь-яке практичне RTT в LAN (~1-5 мс)
Cleanup interval	1 000 мс	Обмеження навантаження на ConcurrentHashMap при потоці 25-30 пакетів/с
UDP recv buffer	8 МБ	Поглинає ~5 кадрів при пікових навантаженнях; 16 МБ призводило до crash на тестових пристроях

2.4. Система просторових анотацій та семантичних маркерів

Постановка задачі та вибір архітектурного підходу базуються на ключовій вимозі до системи V.I.R.A.: щоб анотації, накладені оператором у браузерному інтерфейсі, залишалися прив'язаними до фізичних об'єктів сцени незалежно від подальшого переміщення голови або повороту корпусу користувача у гарнітурі. У підрозділі 1.4 показано, що screen-space анотації (як у TeamViewer Assist AR та PTC

Vuforia Chalk) не задовольняють цю вимогу: вони прив'язані до пікселів відеокадру і зміщуються відносно реальних об'єктів при будь-якому русі камери. World-space підхід, навпаки, розміщує анотацію як 3D-об'єкт у системі координат сцени, що відстежується VI-SLAM-підсистемою гарнітури. V.I.R.A. реалізує саме world-space підхід: кожна анотація (штрих або маркер) отримує тривимірну координату в сцені та прив'язується до неї через OVRSpatialAnchor - просторовий якір, позиція якого стабілізована SLAM-трекером Meta Quest 3S (підрозділ 1.4).

Для структури даних анотації та JSON-протоколу передбачено, що оператор у браузері передає анотації на гарнітуру у вигляді JSON-повідомлень через UDP (порт 50502 у LAN-режимі) або через WebRTC DataChannel (у Cloud-режимі). На стороні Unity кожне вхідне повідомлення десеріалізується у клас ServerAnnotation (лістинг 2.3).

Лістинг 2.3 - Структура даних просторової анотації

```
[Serializable]
public class ServerAnnotation {
    public string id;       // UUID анотації
    public string type;    // "stroke" | "marker" | "flash" | "remove" | "clear" | "freeze" | "unfreeze" | "overlay"
    public float[] points;  // нормалізовані координати точок штриху [x0,y0, x1,y1, ...]
    public string color;    // hex-рядок, наприклад "FF0000"
    public float  width;    // відносна товщина лінії
    public string markerId; // "M1"-"M7"
    public float  normX;    // нормалізована X-координата маркера (0..1)
    public float  normY;    // нормалізована Y-координата маркера (0..1)
    public bool   enabled;  // для типу "overlay": показати/сховати всі анотації
}
```

Поле type визначає гілку обробки у switch-диспетчері AnnotationOverlayManager.ProcessAnnotation(). UUID у полі id дозволяє однозначно

ідентифікувати анотацію протягом усього циклу її існування - від створення до видалення, а також підтримувати ідемпотентне оновлення (повторна передача повідомлення з тим самим id та типом stroke оновлює існуючий штрих, а не створює дублікат).

Центральним алгоритмом підсистеми анотацій для перетворення 2D→3D є метод GetWorldPoint, який забезпечує перетворення нормалізованих екранних координат (normX, normY) у тривимірну точку world-space. Відповідно до аналізу методів визначення глибини у підрозділі 1.4, в системі V.I.R.A. реалізовано двофазний підхід: спочатку застосовується апаратний depth raycast через Meta Environment Depth API, а при його недоступності або промаху - фіксована глибина fallback.

Алгоритм перетворення реалізований у статичному методі GetWorldPoint (лістинг 2.4).

Лістинг 2.4 - Метод перетворення екранних координат у світові (world-space)

```
public static Vector3 GetWorldPoint(
    float normX, float normY,
    PassthroughCameraAccess cameraAccess,
    EnvironmentRaycastManager raycastManager = null,
    Pose? frozenPose = null,
    float fallbackDepth = DEFAULT_FALLBACK_DEPTH)  // = 3.0m
{
    // 1. Y-flip: Passthrough Camera API використовує bottom-left origin
    Vector2 viewportPoint = new Vector2(normX, 1f - normY);
    // 2. Промінь у world-space через оптичні параметри passthrough-камери
    Ray worldRay = cameraAccess.ViewportPointToRay(viewportPoint, frozenPose);
    // 3. Depth raycast через Environment Depth API
    if (raycastManager != null && raycastManager.isActiveAndEnabled) {
        if (raycastManager.Raycast(worldRay, out EnvironmentRaycastHit hit)) {
```

```
            return hit.point;   // hit або HitPointOccluded - обидва валідні
        }
    }
    // 4. Fallback: фіксована глибина вздовж променя
    return worldRay.origin + worldRay.direction * fallbackDepth;
}
```

Крок 1 - Y-flip є обов'язковим: Passthrough Camera API Meta XR SDK використовує систему координат з початком у лівому нижньому куті зображення (bottom-left origin), тоді як браузерний інтерфейс оператора і React Canvas використовують систему з початком у лівому верхньому куті (top-left origin, стандарт HTML). Без Y-інверсії координата normY вказувала б на дзеркально відображений рядок по вертикалі, і анотації розміщувалися б у неправильних точках сцени.

Крок 2 - побудова променя виконується методом PassthroughCameraAccess.ViewportPointToRay(), що враховує реальні оптичні параметри passthrough-камери (фокусні відстані та принципіальну точку з cameraAccess.Intrinsics) та поточну позу голови. Якщо активований режим Freeze Frame (значення frozenPose != null), промінь будується відносно зафіксованої пози, що дозволяє оператору розміщувати анотації на «замороженому» кадрі з точними просторовими координатами без артефактів дрейфу від поточного руху голови.

Крок 3 - depth raycast через EnvironmentRaycastManager.Raycast() є ключовим нововведенням. Meta Environment Depth API надає GPU-текстуру глибини (depth texture) на основі стерео-пасивного бачення гарнітури, яка потім використовується для апаратного raycast-запиту просторової точки без виходу даних у CPU-пам'ять. Як зазначено у підрозділі 1.4, цей підхід є різновидом стереоскопічного визначення глибини, реалізованого безпосередньо на рівні Meta XR SDK, без залежності від зовнішніх ToF-сенсорів або моноскопічної оцінки глибини за однією камерою. Статус EnvironmentRaycastHitStatus.HitPointOccluded - часткове попадання з

оклюзією - також вважається валідним результатом і повертається як точка розміщення.

Крок 4 - fallback з DEFAULT_FALLBACK_DEPTH = 3.0m застосовується, коли EnvironmentRaycastManager недоступний (наприклад, при запуску застосунку до ініціалізації SDK) або raycast не знайшов попадання (наприклад, промінь спрямований у відкритий простір за межами відстаней сканування). Значення 3,0 м є налаштовуваним через Inspector-поле fallbackDepth компонента AnnotationOverlayManager і обране з урахуванням типової відстані до робочої поверхні для задач технічного обслуговування. У першому етапі розробки, до інтеграції Environment Depth API, використовувалось фіксоване значення 1,5 м, яке відповідало середній дистанції витягнутої руки - ситуативно прийнятне, але принципово неточне для об'єктів на різних відстанях (підрозділ 1.4).

Механізм просторових якорів реалізується так: після отримання world-space координати worldPos метод створює GameObject та реєструє його як просторовий якір (лістинг 2.5).

Лістинг 2.5 - Створення та реєстрація просторового якоря

```
public static GameObject CreateAnchor(string id, Vector3 worldPos) {
    var anchorGO = new GameObject($"AnnotationAnchor_{id}");
    anchorGO.transform.position = worldPos;
    anchorGO.AddComponent<OVRSpatialAnchor>();
    return anchorGO;
}
```

Компонент OVRSpatialAnchor з Meta XR SDK реєструє позицію у просторовій карті VI-SLAM підсистеми гарнітури. Це означає, що навіть при фізичному переміщенні користувача по кімнаті або повторному включенні гарнітури (за умови збереження якоря) анотація залишається прив'язаною до того самого місця у просторі - якір «знає» свою позицію відносно feature map SLAM, а не відносно поточної пози голови. Як зазначено у підрозділі 1.4, просторовий дрейф

VI-SLAM у Meta Quest 3 після loop closure не перевищує 0,5-8 см на 10 м переміщення, що є прийнятним для задач технічного супроводження. Усі анотації (штрихи та маркери) є дочірніми об'єктами (SetParent) відповідного OVRSpatialAnchor-GameObject, що забезпечує єдиний SLAM-tracked transform для кожної анотації.

Серед типів штрихових анотацій базовим є штрих (тип stroke), що реалізується через LineRenderer з useWorldSpace = true. Метод BuildWorldPositions() перетворює весь масив нормалізованих точок штриху у масив world-space координат, викликаючи GetWorldPoint() для кожної пари (normX, normY). LineRenderer налаштовується наступним чином:

- startColor / endColor - колір з поля color (HTML hex);
- startWidth / endWidth - 0.005f * Mathf.Max(1f, width / 3f) метра, тобто ~5 мм базова товщина з масштабуванням відносно значення width;
- numCapVertices = 5, numCornerVertices = 5 - скруглені кінці та кути для природного вигляду;
- матеріал - Sprites/Default (без освітлення, Alpha Blend).

Окремим типом є семантичні маркери M1-M7. На відміну від штрихів, семантичний маркер є тривимірним білбордним спрайтом із фіксованим семантичним значенням. Система містить 7 типів маркерів, зареєстрованих у ScriptableObject MarkerRegistry (табл. 2.4).

Таблиця 2.4 - Типи семантичних маркерів системи V.I.R.A.

ID	Колір (hex)	Семантичне значення
M1	#ffbd28 (жовтий)	Увага / небезпека (ISO W001)
M2	#f41f01 (червоний)	Критична несправність / стоп
M3	#3128ff (синій)	Вказівка на місце / орієнтир
M4	#41aa2a (зелений)	Підтвердження / правильно
M5	#b9b9b9 (сірий)	Запитання / уточнення
M6	#01de43 (яскраво-зелений)	Готово / OK
M7	#ff9e29 (помаранчевий)	Загальна анотація / позначка

Кожен маркер будується з PNG-спрайту розміром 512×512 пікселів та компонента MarkerBehaviour. Спрайти стискаються у форматі ETC2_RGBA8 при компіляції під Android/Quest, що забезпечує апаратне декодування на GPU Adreno 740. Клас MarkerPrefabBuilder (Editor-інструмент) автоматично генерує .prefab-файли з прив'язаним спрайтом, кольором і компонентом MarkerBehaviour, а також реєструє їх у MarkerRegistry.

Поведінка маркера визначається класом MarkerBehaviour, який реалізує повний набір поведінок world-space маркера у кожному кадрі Update():

- Циліндричний білборд (CylindricalBillboard): маркер постійно повертається по осі Y у бік камери за формулою atan2(dx, dz) у площині XZ, що забезпечує читабельність піктограми під будь-яким горизонтальним кутом перегляду без нахилу спрайту по вертикалі. Режим SnapToNearestCardinal додатково квантизує поворот до найближчого кардинального напряму (0°, 90°, 180°, 270°) з гістерезисом hysteresisAngle = 20°, що знижує «дрижання» при перегляді маркера під кутом ~45°.
- Вертикальне floating: маркер розташовується на висоті heightOffset = 0.25m над SLAM-якорем, причому ця висота зменшується за експоненційним законом при наближенні до якоря: heightT = linearT^lineCurve де lineCurve = 2.5 - коефіцієнт кривої. При відстані понад lineFullHeightDist = 2.0m маркер зупиняється на повній висоті.
- Proximity fade: якщо відстань від камери до якоря менша за fadeStartDist = 0.60m, прозорість маркера поступово знижується до minAlpha = 0.50 при досягненні fadeEndDist = 0.30m, запобігаючи «залипанню» спрайту в обличчя при дуже близькому підході.
- Масштаб із обмеженням: localScale = worldSize * clamp(1, minScale, maxScale), де worldSize = 0.25m. Це гарантує, що маркер залишається видимим з будь-якої відстані (мінімум 5 см) і не стає надмірно великим (максимум 300% від worldSize).
- Anchor line: LineRenderer з двома точками - від позиції SLAM-якоря до позиції маркера - візуально з'єднує маркер із точкою на поверхні. Прозорість лінії

heightT * 0.5 визначається тією самою експоненційною кривою, що і вертикальне floating: лінія зникає одночасно із «приземленням» маркера.

- Flash анімація (FlashPulse()): синусоїдальний спалах на 3 секунди з формулою pulse = sin(t * π * 7 * (1 - t)) * 0.8 - сім затухаючих пульсацій, що сигналізують про нещодавнє розміщення або оновлення маркера. Спалах є результатом виклику команди type: "flash" від оператора або автоматично при створенні маркера.

Для синхронізації між Quest і браузером використовується механізм Freeze Frame, який вирішує проблему точного позиціонування анотацій: коли оператор зупиняє кадр у браузері (команда type: "freeze"), AnnotationOverlayManager зберігає поточну позу камери (лістинг 2.6).

Лістинг 2.6 - Обробка команди фіксації кадру (Freeze Frame)

```
private void HandleFreeze() {
    frozenPose = cameraAccess.GetCameraPose();
}
```

Поки frozenPose != null, всі подальші виклики GetWorldPoint() будують промінь відносно зафіксованої пози, а не поточної. Це дозволяє оператору бачити нерухомий кадр і розміщувати анотації, тоді як сам користувач з гарнітурою може продовжувати рухатися - анотації все одно потраплять у правильні world-space точки, що відповідають зафіксованому моменту часу. Команда type: "unfreeze" скидає frozenPose = null, повертаючи систему до live-режиму.

Управління активними анотаціями ведеться через словник Dictionary<string, GameObject> activeAnnotations, де ключ - це UUID анотації з поля id. Видалення конкретної анотації (type: "remove" з відповідним id) відбувається через Destroy(anchorGO) та видалення запису зі словника. Команда type: "clear" знищує всі анотації та скидає frozenPose. Команда type: "overlay" з полем enabled дозволяє тимчасово приховати всі анотації без їх видалення - корисно при огляді сцени без візуального «засмічення».

2.5. Android Relay: мостовий компонент між UDP та WebSocket

У контексті обґрунтування необхідності компонента варто зазначити, що гарнітура Meta Quest 3S є standalone Android-пристроєм, яка функціонує у Wi-Fi-мережі та не підтримує пряме встановлення WebRTC-з'єднання з браузером оператора з причин, детально описаних у підрозділі 2.2. Однак проблема не зводиться лише до відсутності бібліотечної підтримки WebRTC у Unity/Quest: навіть якщо б таке з'єднання було технічно можливим, Quest-гарнітура за типових умов розгортання перебуває за NAT корпоративного або приватного Wi-Fi маршрутизатора і не має публічної IP-адреси. Як зазначено у підрозділі 1.3, протокол ICE (Interactive Connectivity Establishment) виконує NAT-traversal через STUN-сервер для отримання server-reflexive адреси або через TURN-сервер для relay-передачі у разі симетричного NAT; проте ICE і RTCPeerConnection є API браузерного середовища та потребують JavaScript або еквівалентного runtime, який відсутній на платформі Unity/IL2CPP для Quest. Таким чином, Android Relay вирішує одночасно три архітектурні проблеми:

1. Протокольний міст: гарнітура надсилає UDP, хмарний сервер очікує WebSocket - Android Relay виконує перетворення між цими двома транспортами.
2. NAT-traversal: Android-смартфон, на відміну від гарнітури, підтримує повноцінний WebRTC-стек через стандартний RTCPeerConnection, а WebSocket-з'єднання до хмарного сервера ініціює relay-пристрій, що дозволяє обійти NAT без ICE на боці Quest.
3. Мережева ізоляція гарнітури: Quest-гарнітура залишається в локальній мережі і не потребує прямого виходу в Інтернет - вся хмарна взаємодія відбувається через Android Relay як єдину точку виходу.

Загальна архітектура застосунку Android Relay базується на мові Kotlin (Android API 26+) із використанням kotlinx.coroutines для асинхронного вводу-виводу та OkHttp 4.12.0 для WebSocket-з'єднання. Застосунок складається з восьми

класів з чіткою відповідальністю. Структуру та відповідальність класів застосунку наведено у табл. 2.5.

Таблиця 2.5 - Структура та відповідальність класів модуля Android Relay

Клас	Відповідальність
MainActivity	UI, відображення стану, введення URL хмарного сервера
RelayService	Android Foreground Service; утримує lifecycle компонентів при згорнутому UI
RelayOrchestrator	Центральний координатор: з'єднує всі компоненти, управляє lifecycle
DiscoveryBroadcaster	UDP broadcast 255.255.255.255:50500 кожні 2 с (пошук гарнітури)
VideoReceiver	UDP listener на порту 50501, буфер 8 МБ
FrameAssembler	Збирання JPEG-кадрів з UDP-фрагментів (детально - підрозділ 2.3)
AnnotationForwarder	UDP sender на порт 50502; перенаправляє JSON-анотації від сервера до Quest
CloudWebSocket	WebSocket-клієнт (OkHttp); бінарна та текстова передача до хмарного сервера
NetworkUtils	Визначення IP-адреси та broadcast-адреси активного Wi-Fi інтерфейсу

Модель потоків і корутин у розробленому модулі побудована на принципі розмежування обчислювального навантаження між потоками «не виконувати IO у main thread»:

- IO Thread (Dispatchers.IO): VideoReceiver (блокуючий DatagramSocket.receive()), DiscoveryBroadcaster (циклічний DatagramSocket.send() кожні 2 с);
- OkHttp internal thread: CloudWebSocket (управляється OkHttp автоматично; WebSocket read/write виконуються в окремому пулі потоків);
- CPU-bound, без прив'язки: FrameAssembler (асемблювання фрагментів у ConcurrentHashMap, конкатенація байтових масивів - процесороємна операція без блокування сокета);

- Main Thread: MainActivity (оновлення UI-стану через StateFlow, зібраний repeatOnLifecycle для запобігання витокам lifecycle-coroutine).

Таке розмежування гарантує, що затримка UDP-прийому (VideoReceiver) не залежить від швидкості WebSocket-відправлення (CloudWebSocket): між ними FrameAssembler виступає буфером і природно поглинає різницю у швидкостях.

Опис потоку даних через RelayOrchestrator дозволяє зрозуміти роль цього центрального класу, що з'єднує всі компоненти в єдину систему, є RelayOrchestrator. Після виклику orchestrator.start(cloudUrl) встановлюються з'єднання між компонентами, зображені на рисунку 2.1.

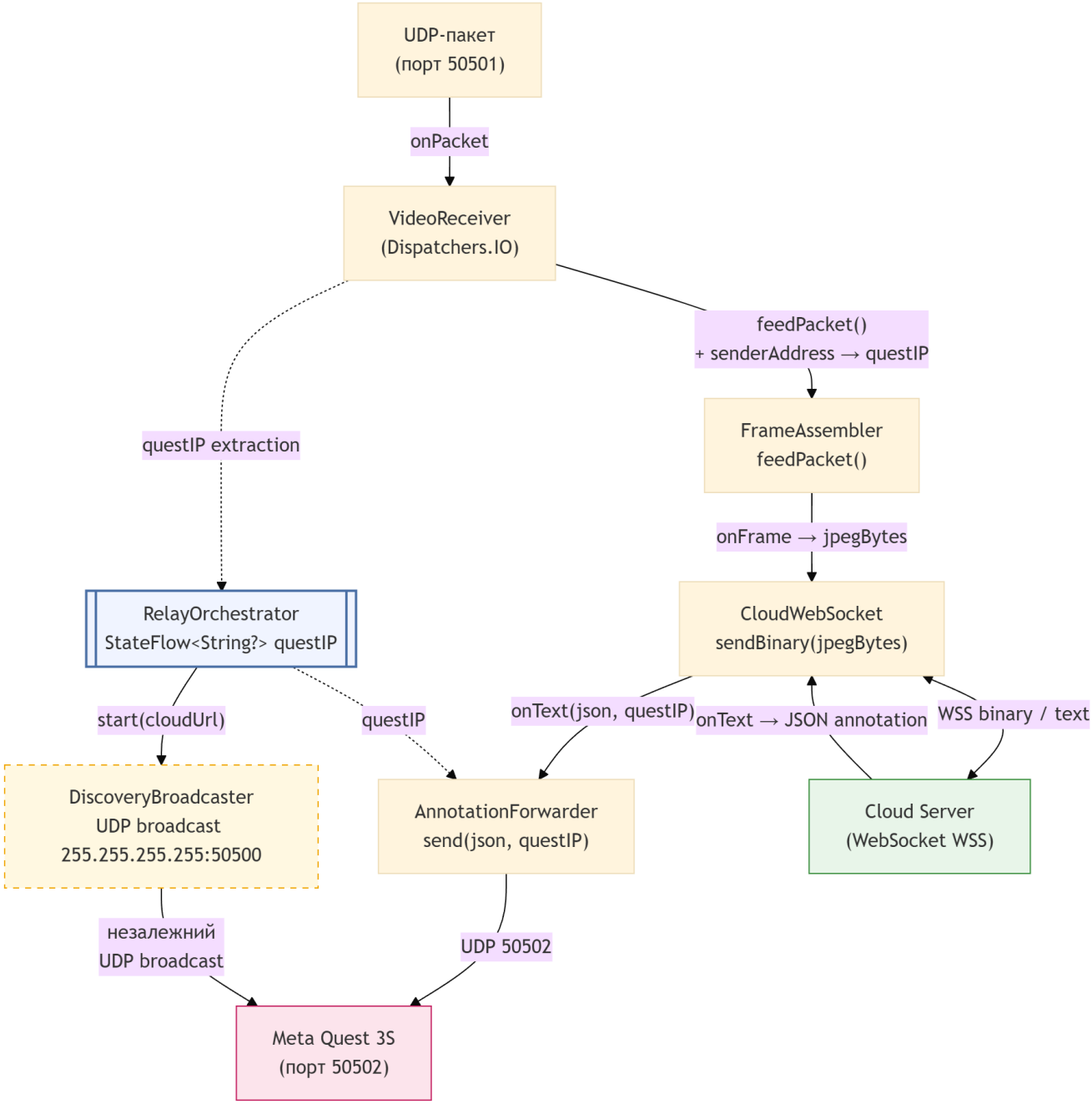

Рисунок 2.1 - Діаграма потоку даних через компонент RelayOrchestrator у модулі Android Relay

Виявлення IP-адреси гарнітури відбувається пасивно: VideoReceiver витягує senderAddress з першого ж отриманого UDP-пакету на порту 50501 і передає його в RelayOrchestrator, який зберігає значення у StateFlow<String?> questIP. Цей IP потім використовує AnnotationForwarder для зворотного UDP-відправлення анотацій. Таким чином, явна конфігурація IP-адреси гарнітури не потрібна - вона визначається автоматично.

WebSocket-канал між Android Relay та хмарним сервером є мультиплексованим: по ньому паралельно передаються бінарні та текстові повідомлення з різним семантичним значенням:

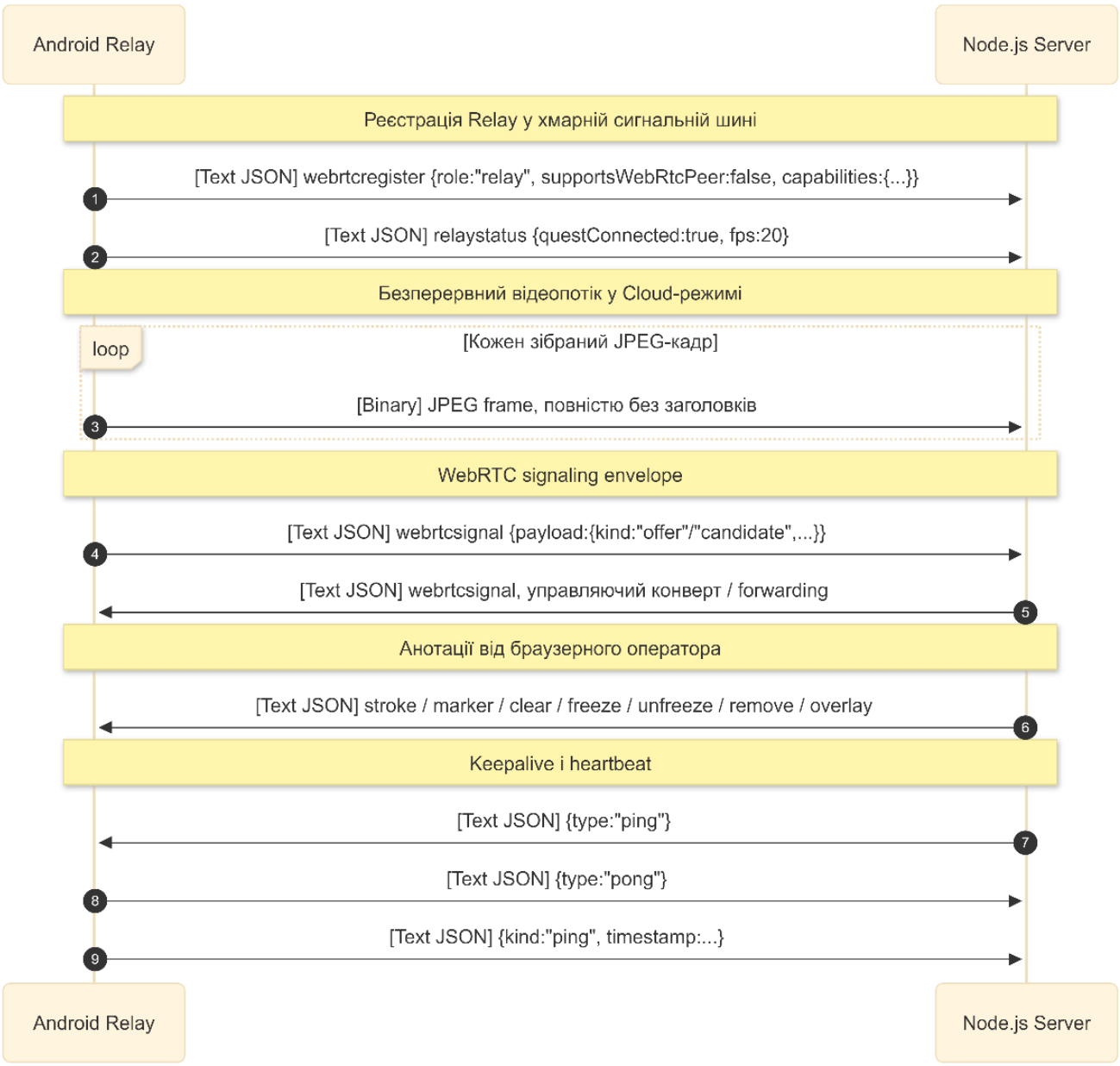

Рисунок 2.2 - Діаграма послідовності повідомлень WebSocket-каналу між модулем Android Relay та хмарним сервером Node.js

На рисунку 2.2 зображено послідовність повідомлень, що передаються по мультиплексованому WebSocket-каналу між Android Relay та хмарним сервером. Канал обслуговує чотири логічно відокремлені потоки: реєстрацію Relay у

сигнальній шині (повідомлення webrtcregister, relaystatus); безперервний відеопотік у вигляді бінарних JPEG-кадрів без заголовків; WebRTC-сигналізацію через конверт webrtcsignal з корисним навантаженням типів offer/candidate; анотації від браузерного оператора з типами stroke, marker, clear, freeze, unfreeze, remove, overlay. Окремо виділено механізм підтримки з'єднання: Plain keepalive (ping/pong) ініціюється сервером, тоді як Envelope heartbeat (kind:"ping") із мітками часу надсилається самим Relay для вимірювання RTT.

Фільтрація вхідних повідомлень у CloudWebSocket.onTextMessage виконується за полем type: повідомлення анотаційних типів передаються до AnnotationForwarder для пересилання на Quest; повідомлення типів ping/pong та webrtcsignal обробляються власне Relay як управляюча шина (control-path) і не пересилаються на гарнітуру.

Процес реєстрації Relay як WebRTC-учасника відбувається під час підключення до хмарного сервера через надсилання конверту webrtcregister з полем role:"relay" та об'єктом capabilities, що містить supportsWebRtcPeer: false. Це поле сигналізує серверу і браузерному інтерфейсу, що Relay є «signaling-only» peer: він бере участь у WebRTC-сигналінговій шині для обміну SDP offer/answer та ICE-кандидатами, але не встановлює самостійну RTCPeerConnection як кінцеву точку відео. Натомість відеопотік у Cloud-режимі передається від Relay до сервера по WebSocket у вигляді бінарних JPEG-фреймів, а вже сервер ретранслює їх браузеру по WebRTC (детально - підрозділ 2.6). Це рішення дозволяє поступово мігрувати від WebSocket-relay до повноцінного WebRTC без розриву зворотної сумісності.

Механізм heartbeat і keepalive у системі забезпечує підтримку довготривалого WebSocket-з'єднання між Relay та хмарним сервером підтримується за допомогою двох механізмів:

- Plain ping/pong: Relay відповідає на вхідний {type:"ping"} відповіддю {type:"pong"}. Це забезпечує базову перевірку живості з'єднання на рівні застосунку.

- Envelope heartbeat: Relay самостійно надсилає конверт {kind:"ping", timestamp:...} кожні 10 секунд. Якщо сервер не підтвердив heartbeat протягом 30 секунд - з'єднання вважається обірваним і запускається reconnect.

Щодо обробки помилок і стратегії reconnect, слід зазначити, що всі можливі сценарії відключення реалізовані в RelayOrchestrator:

- Втрата з'єднання з Quest: якщо UDP-пакети від гарнітури не надходять протягом 5 секунд, questConnected переходить у false, UI відображає «Quest Disconnected», DiscoveryBroadcaster продовжує роботу для повторного виявлення.
- Втрата WebSocket до Cloud: при будь-якій помилці CloudWebSocket (onFailure, onClosed) запускається механізм exponential backoff reconnect з інтервалами 1 → 2 → 4 → 8 → 16 → 30 с (cap = 30 с). Під час очікування reconnect DiscoveryBroadcaster та VideoReceiver продовжують роботу: пакети від Quest збираються і відкидаються (оскільки CloudWebSocket.isConnected == false), але позиція гарнітури та questIP зберігаються.
- questIP == null при відправленні анотацій: захист від NullPointerException - AnnotationForwarder.send() перевіряє questIP != null перед UDP-відправленням і логує попередження без краш-у.
- Переповнення UDP recv buffer: VideoReceiver з буфером 8 МБ (~5 кадрів) при затримці Cloud-відправлення; FrameAssembler.staleCleanup() превентивно видаляє буфери, що очікують понад 500 мс, знижуючи backpressure.
- Некоректний chunkIndex: FrameAssembler ігнорує пакет з chunkIndex >= totalChunks (захист від некоректних або навмисно спотворених пакетів без краш-у).

Використання Foreground Service та Wi-Fi Lock дозволяє гарантувати стабільну роботу relay-з'єднання в умовах фонового режиму Android-застосунку реалізовано два механізми з Post-MVP функціоналу:

- RelayService як Foreground Service: утримує процес застосунку від переведення Android у стан deep sleep з видаленням з пам'яті при згортанні UI. Foreground Service відображає постійне сповіщення «V.I.R.A. Relay active» у панелі

сповіщень, що є обов'язковою вимогою Android API 26+ для довготривалих фонових процесів (дозвіл FOREGROUND_SERVICE).

- Wi-Fi Lock (CHANGE_WIFI_MULTICAST_STATE): утримує активним Wi-Fi модуль без переходу у режим збереження енергії, що запобігає підвищенню затримки UDP-прийому при агресивному power management адаптера Wi-Fi деяких Android-пристроїв.

Аналіз нефункціональних характеристик та підтверджених показників за результатами польових випробувань дозволив зафіксувати наступні показники продуктивності Android Relay:

- Середній FPS передачі (avg throughput): 18,39 кадрів/с;
- Коефіцієнт підключення Quest/Cloud: 100% (без розривів протягом 30 хв);
- Використання RAM: не більше 128 МБ (обмеження платформи дотримано);
- Навантаження на CPU: не більше 15% при потоці 25 FPS;
- Усі 36 unit-тестів (FrameAssembler, Discovery, CloudWebSocket, Integration) та fault-injection сценарії пройшли успішно (BUILD SUCCESSFUL).

2.6. Серверна інфраструктура: Node.js, WebSocket-relay та coturn

Для обґрунтування self-hosted підходу слід звернутися до аналізу моделі загроз у підрозділі 1.6, де визначено три ключові атрибути безпеки: конфіденційність (confidentiality), цілісність (integrity) та доступність (availability). Там само зазначено, що relay-архітектура вносить специфічний ризик: relay-вузол, через який проходить медіапотік у відкритому вигляді, є потенційною точкою компрометації даних - і якщо цей вузол знаходиться під управлінням стороннього SaaS-провайдера, організація-оператор втрачає контроль над конфіденційністю відеоданих з виробничого середовища (підрозділ 1.6). Аналіз комерційних рішень

у підрозділі 1.1 підтверджує: усі розглянуті конкуренти (MS Dynamics 365, TeamViewer Assist AR, PTC Vuforia Chalk, Scope AR) є SaaS-системами з обов'язковою передачею медіапотоку через хмарну інфраструктуру постачальника, що унеможливлює їх застосування у закритих мережах або при роботі з конфіденційними об'єктами (підрозділ 1.1). Self-hosted Node.js сервер та coturn у V.I.R.A. вирішують обидві проблеми: оператор розгортає інфраструктуру на власному (або орендованому) ресурсі, зберігаючи повний контроль над трафіком, ключами та конфігурацією безпеки.

Структура Node.js серверного застосунку базується на модульній архітектурі та реалізована з використанням Node.js + TypeScript з мінімальним набором залежностей і складається з чотирьох модулів. Структуру модулів серверного бекенду наведено у табл. 2.6.

Таблиця 2.6 - Структура модулів серверного застосунку Node.js

Модуль	Файл	Відповідальність
Головний сервер	index.ts	Express HTTP + ws WebSocket сервер; монтування маршрутів; статична роздача фронтенду
UDP-приймач відео	udp-receiver.ts	Прийом JPEG-фрагментів від Quest (LAN) або JPEG-фреймів від Android Relay (Cloud) на порту 50501
UDP-відправник анотацій	udp-sender.ts	Відправлення JSON-анотацій на Quest по UDP порт 50502; зберігає IP-адресу Quest
WebSocket-обробник	websocket.ts	Управління підключеними браузерними клієнтами; розподіл відеофреймів; WebRTC-сигналінг

Вибір Node.js як платформи обґрунтований моделлю виконання на основі event loop: сервер є relay-вузлом з переважно I/O-bound навантаженням (прийом UDP → відправлення WebSocket, без CPU-важкого кодування), де асинхронна однопотокова модель Node.js показує вищу пропускну здатність при меншому споживанні ресурсів порівняно з multi-thread архітектурами. На платформі Hetzner

CAX11 (ARM, 2 vCPU, 4 ГБ RAM) сервер стабільно обробляє потік ~18-25 JPEG-фреймів/с (~1,5 МБ/с) без перевантаження event loop.

Змінні середовища для конфігурації сервера наведено у табл. 2.7.

Таблиця 2.7 - Змінні середовища серверної інфраструктури

Змінна	Значення за замовчуванням / допустимі	Призначення
VIDEOPORT	50501	UDP-вхід: відеокадри від Quest (LAN) або Android Relay (Cloud)
ANNOTATIONPORT	50502	UDP-вихід: JSON-анотації до Quest
WEBPORT	5000	HTTP/WebSocket: браузерний клієнт і сигналінг
VIRA_TRANSPORT_MODE	local|udpws / cloud|webrtc	Режим транспортного рівня
VIRA_DISCOVERY_ENABLED	true / false	LAN broadcast discovery
VIRA_WEBRTC_SIGNALING_ENABLED	true / false	Killswitch WebRTC-сигналінгу
VIRA_ICE_SERVERS_JSON	[{"urls":"turn:..."}]	TURN/STUN конфігурація у форматі JSON
VIRA_ICE_TRANSPORT_POLICY	all / relay	Примусове використання TURN relay

Розділення VIDEOPORT та ANNOTATIONPORT на окремі UDP-сокети є обдуманим рішенням: воно дозволяє незалежно моніторити або перенаправляти два напрямки трафіку, не ускладнюючи логіку мультиплексування.

Розглядаючи функції модуля websocket.ts, слід виділити дві незалежні ролі, які він виконує: relay відеопотоку та сигналінг

Роль 1 - Binary relay. При отриманні бінарного WebSocket-повідомлення від Android Relay (JPEG-кадр) сервер розсилає його broadcast-ом до всіх підключених браузерних клієнтів. У LAN-режимі аналогічну роль виконує udp-receiver.ts: при отриманні зібраного JPEG-кадру від Quest по UDP він передає його у websocket.ts для розсилки. Таким чином, модуль відео-relay є спільним для обох транспортних режимів - відмінність лише у джерелі фреймів (UDP сокет у LAN або WebSocket від Android Relay у Cloud).

Роль 2 - WebRTC signaling relay. При отриманні текстового JSON-повідомлення з полем type: "webrtcregister" або type: "webrtcsignal" сервер виступає сигналінговим брокером між Android Relay та браузером. Алгоритм обробки:

- webrtcregister: клієнт (Relay або браузер) реєструється у peer-директорії сервера з полями clientId, role та capabilities; сервер повертає webrtcregistered з поточним списком peers;
- webrtcsignal: конверт з targetClientId маршрутизується до конкретного peer у директорії; payload містить SDP offer/answer або ICE-кандидат і не інтерпретується сервером - він є прозорим для нього;
- webrtcerror: надсилається клієнту при помилці маршрутизації (невідомий targetClientId);
- Feature flag VIRA_WEBRTC_SIGNALING_ENABLED = false деактивує обробники webrtcregister/webrtcsignal без зупинки сервера - killswitch для поступового введення WebRTC в продакшн.

Relay-observability реалізована через текстові status-повідомлення від Android Relay: {type:"relaystatus", questConnected:true, fps:20, droppedFrames:0, invalidPackets:0} - сервер передає їх браузерному клієнту для відображення у debug sidebar.

Механізм автоматичного визначення IP Quest у LAN-режимі дозволяє серверу функціонувати без ручного налаштування адрес гарнітури: udp-receiver.ts витягує remoteInfo.address з першого отриманого UDP-пакету і передає значення у udp-

sender.ts для встановлення цільової адреси відправлення анотацій. При надходженні UDP-пакетів від нового IP (перепідключення Quest після перезапуску) адреса оновлюється автоматично.

Щодо конфігурації та розгортання coturn, слід зазначити, що цей TURN-сервер розміщується на тому самому хості Hetzner CAX11, де знаходиться Node.js backend, та виконує роль relay-вузла для WebRTC ICE при NAT-traversal між Android Relay (ініціатором RTCPeerConnection) та браузером оператора (підрозділ 1.3). Базова конфігурація TURN-сервера наведена у лістингу 2.7.

Лістинг 2.7 - Базова конфігурація TURN-сервера coturn

```
apt install coturn
# /etc/turnserver.conf:
listening-port=3478          # стандартний TURN/STUN порт
tls-listening-port=5349      # TURN over TLS
relay-ip=<публічний IPv4>    # IP хоста Hetzner CAX11
realm=vira.example           # realm для автентифікації
user=vira:<shared_secret>    # HMAC-SHA1 credential
lt-cred-mech                 # Long-Term Credential Mechanism (RFC 5389)
```

Вибір coturn обумовлений такими аргументами: coturn є відкритим програмним забезпеченням (BSD-ліцензія) з підтримкою всіх необхідних механізмів RFC 5766 (TURN), RFC 5389 (STUN) та RFC 8656 (TURN over TLS/DTLS); він є де-факто стандартом для self-hosted WebRTC TURN у відкритих проєктах; розгортання зводиться до однієї команди apt install coturn на Ubuntu/Debian, що відповідає вимозі мінімальної операційної складності.

Для запобігання несанкціонованому використанню TURN-сервера реалізовано механізм HMAC-SHA1 time-limited credentials відповідно до RFC 5389: сервер Node.js генерує тимчасовий credential з TTL, після закінчення якого coturn відхиляє нові allocation-запити з цим credential. Це відповідає рекомендаціям підрозділу 1.6 щодо захисту від DoS через TURN-credentials: не зберігається

статичний пароль у клієнтському коді - браузер отримує credential через VIRA_ICE_SERVERS_JSON у runtime з обмеженим терміном дії.

Транспортна безпека серверного рівня на хмарному розгортанні забезпечується застосуванням багаторівневого захисту відповідно до моделі загроз підрозділу 1.6:

- Reverse proxy (Nginx/Caddy) з TLS 1.3-сертифікатом (Let's Encrypt): термінує HTTPS/WSS-з'єднання перед Node.js backend; WebSocket-з'єднання Android Relay та браузера здійснюються виключно по wss://, що шифрує SDP-конверти та ICE-кандидати (захист від MITM, описаного у підрозділі 1.6);
- UFW firewall: відкриті лише порти 443 (HTTPS/WSS), 3478 (TURN/STUN UDP+TCP), 5349 (TURNS/TLS), 22 (SSH); порт 50501/UDP закритий для прямого доступу з Інтернету (прийом відео лише через WebSocket від Android Relay у Cloud-режимі);
- systemd unit: Node.js backend та coturn запускаються як системні сервіси з Restart=on-failure та RestartSec=5s, що забезпечує автоматичне відновлення після краш-у без ручного втручання.

Щодо фронтенду, його статична роздача та prod build реалізовані через збірку проєкту командою vite build в статичний bundle та роздається безпосередньо Express-сервером (порт 5000) як статичні файли - окремий веб-сервер для фронтенду не потрібен. У режимі розробки використовується vite dev server на порту 5173 з proxy WebSocket-підключень до backend на порту 5000. Це спрощує розгортання до єдиного процесу Node.js, що особливо важливо для обмеженої конфігурації Hetzner CAX11 (2 vCPU, 4 ГБ RAM, вартість ~3,79 €/міс). Перелік задіяних мережевих портів наведено у табл. 2.8.

Таблиця 2.8 - Мережеві порти серверної інфраструктури

Порт	Протокол	Напрямок	Призначення
443	TCP (WSS/HTTPS)	In	WebSocket від Android Relay і браузера; HTTPS фронтенд (через reverse proxy)
5000	TCP (HTTP/WS)	In	Node.js backend (dev/LAN); пряме WebSocket-підключення без TLS

Порт	Протокол	Напрямок	Призначення
50501	UDP	In	Відеофрейми від Quest (тільки LAN-режим, внутрішня мережа)
50502	UDP	Out	JSON-анотації до Quest (тільки LAN-режим)
3478	UDP + TCP	In	coturn TURN/STUN (WebRTC ICE allocation)
5349	TCP (TLS)	In	coturn TURNS over TLS (шифрований TURN-relay)

2.7. Браузерний інтерфейс оператора: React/TypeScript та WebRTC-сесія

У контексті обґрунтування вибору платформи слід зазначити, що браузер обраний як середовище виконання інтерфейсу оператора з трьох причин: по-перше, він є єдиним стандартним runtime, що надає нативний доступ до RTCPeerConnection API без встановлення додаткового програмного забезпечення (підрозділ 1.3); по-друге, браузерний інтерфейс не потребує інсталяції і доступний з будь-якого пристрою оператора - ПК, ноутбука або планшета; по-третє, браузер надає HTMLCanvasElement як примітив для накладення растрової та векторної графіки поверх відеопотоку без звернення до зовнішніх UI-фреймворків для рендерингу. Вибір React + TypeScript + Vite як технологічного стеку обумовлений типобезпечністю TypeScript при роботі з WebRTC-конвертами складної структури, реактивною моделлю стану для синхронізації багатьох джерел подій (WebSocket, RTCPeerConnection, UI) та швидким HMR-циклом розробки через Vite.

Компонентна структура застосунку включає ряд основних елементів фронтенду, що наведені у табл. 2.9.

Таблиця 2.9 - Структура компонентів браузерного інтерфейсу оператора

Файл	Тип	Відповідальність
App.tsx	React-компонент	Головний layout; управління станом з'єднання; bootstrap WebRTC-сигналінгу; debug sidebar
VideoCanvas.tsx	React-компонент	Відображення відеопотоку на <canvas>; малювання штрихів; Freeze Frame; drag-and-drop маркерів; AR overlay toggle
Controls.tsx	React-компонент	Панель управління: вибір інструменту, колір, товщина лінії, кнопки Freeze/Clear/Theme/Fullscreen
MarkerPalette.tsx	React-компонент	Палітра семантичних маркерів M1-M7 у вигляді SVG-піктограм; drag-and-drop джерело
network/viraSocket.ts	TypeScript-модуль	Типізований WebSocket/сигналінговий хелпер; парсинг конвертів; dispatch подій
network/webrtcSession.ts	TypeScript-модуль	Scaffold RTCPeerConnection; управління lifecycle SDP offer/answer/ICE; стан сесії
SoundManager.ts	TypeScript-модуль	Аудіозворотний зв'язок: button.wav, clear.wav, videoonline.wav, videooffline.wav
translations.ts	TypeScript-модуль	Інтернаціоналізація: EN, UA, DE, PL, PT, JP

Для відображення відеопотоку обрано використання елемента canvas замість video, оскільки потік доставляється у формі JPEG-фреймів по WebSocket (LAN-режим) або по WebRTC DataChannel (Cloud-режим). Тому стандартний HTMLVideoElement не застосовний - він призначений для декодування container-форматів (MP4, WebM) і не підтримує pushFrame()-інтерфейс для окремих JPEG-блобів. Натомість VideoCanvas.tsx реалізує таку схему рендерингу:

1. При отриманні бінарного WebSocket-повідомлення (або бінарного DataChannel-повідомлення у Cloud-режимі) байти JPEG передаються у URL.createObjectURL(new Blob([data], {type:'image/jpeg'})).

2. Створюється HTMLImageElement, для якого встановлюється src = objectURL.

3. По події image.onload виконується ctx.drawImage(image, 0, 0, canvas.width, canvas.height) з масштабуванням із збереженням пропорцій (letterboxing) - canvas заповнює максимальну область контейнера, а зображення центрується з чорними полями.

4. Після drawImage об'єктний URL звільняється через URL.revokeObjectURL(objectURL) для запобігання витоку пам'яті.

Такий підхід надає повний контроль над рендерингом кожного кадру і дозволяє у тому самому Update-циклі накладати анотаційний шар поверх відеозображення засобами Canvas 2D API.

Алгоритм малювання штрихів і нормалізація координат базуються на подіях pointerdown, де кожна точка переміщення курсора перетворюється у нормалізовані координати (лістинг 2.8).

Лістинг 2.8 - Алгоритм нормалізації координат курсора відносно відеокадру

```
const rect = canvas.getBoundingClientRect();
// letterbox-скоригована область відео всередині canvas
const videoRect = getVideoRect(canvas, videoAspectRatio); // враховує letterboxing
const normX = (event.clientX - rect.left - videoRect.x) / videoRect.width;
const normY = (event.clientY - rect.top  - videoRect.y) / videoRect.height;
// normX, normY ∈ [0.0, 1.0] - відносно меж відеозображення, не canvas
```

Нормалізація відносно videoRect, а не canvas.getBoundingClientRect(), є принципово важливою: відеозображення займає не весь canvas (letterboxing залишає поля), і якби нормалізація виконувалась відносно всього canvas-у, координати поза межами відео давали б значення, що виходять за, і Quest розміщував би анотації поза полем зору. Після завершення штриху (pointerup) накопичений масив нормалізованих точок [x0, y0, x1, y1, ...] відправляється на сервер у форматі JSON (лістинг 2.9).

Лістинг 2.9 - Формат команди створення штрихової анотації

```
{
  "type": "stroke",
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "points": [0.12, 0.34, 0.15, 0.37, 0.18, 0.40],
  "color": "FF0000",
  "width": 3.0
}
```

Функція Freeze Frame у браузері реалізована через кнопку Freeze у модулі Controls.tsx, що надсилає відповідну команду {type:"freeze"} і встановлює прапорець isFreezed = true у стані VideoCanvas.tsx. У замороженому стані drawImage не викликається при надходженні нових кадрів - останній отриманий JPEG залишається на канвасі нерухомим. Оператор малює штрихи поверх замороженого кадру, і всі анотації прив'язуються до world-space позицій відносно пози камери в момент заморожування (детально - підрозділ 2.4). При натисканні Unfreeze надсилається {type:"unfreeze"}, прапорець скидається, відновлюється live-відображення.

Механізм drag-and-drop семантичних маркерів реалізований через компонент MarkerPalette.tsx, що дозволяє оператору перетягувати піктограми на відеоканвас: при подіях dragstart встановлюється dataTransfer.setData("markerId", "M1"), при drop на <canvas> обчислюються нормалізовані координати точки скидання і відправляється відповідна команда (лістинг 2.10).

Лістинг 2.10 - Формат команди розміщення семантичного маркера

```
{
  "type": "marker",
  "id": "uuid-...",
  "markerId": "M1",
  "normX": 0.47,
```

```
  "normY": 0.61
}
```

Піктограма маркера відображається поверх відеоканваса у точці скидання як тимчасовий 2D-оверлей до надходження підтвердження від Quest. Команда {type:"flash", "id":"uuid-..."} може бути надіслана повторним кліком на вже розміщений маркер у списку активних анотацій для привернення уваги користувача у гарнітурі.

Модуль webrtcSession.ts реалізує scaffold RTCPeerConnection відповідно до стандартного SDP offer/answer handshake, описаного у підрозділі 1.3. Браузер є initiator (Caller) у WebRTC-сесії; Android Relay є responder (Callee):

Сигналінгові конверти webrtcsignal передаються через WebSocket-канал до сервера (підрозділ 2.6), який маршрутизує їх до цільового peer за targetClientId. ICE-конфігурація завантажується динамічно з VITE_VIRA_ICE_SERVERS_JSON (Vite env-змінна), що дозволяє змінювати STUN/TURN параметри без перекомпіляції фронтенду.

Управління lifecycle WebRTC-сесії реалізовано у модулі webrtcSession.ts через створення об'єкта RTCPeerConnection і реалізує повний lifecycle з обробкою відмов:

- iceconnectionstate = "failed" → автоматичний restartIce() та повторний offer;
- signalingState = "closed" або WebSocket disconnect → скидання RTCPeerConnection, повернення у стан реєстрації (webrtcregister);
- Exponential backoff retry при повторному підключенні (аналогічно до механізму, описаного у підрозділі 2.5 для Android Relay);
- Debug sidebar у App.tsx відображає поточний стан: pcState (new/connecting/connected/disconnected/failed), iceState (checking/connected/completed), signalingState, лічильники tx/rx байт через DataChannel.

Використання feature flags для поступового введення дозволяє фронтенду контролювати розгортання WebRTC-стеку:

- VITE_VIRA_WEBRTC_ENABLED = false - повністю вимикає webrtcSession.ts; застосунок працює в legacy WebSocket-режимі (LAN);
- VITE_VIRA_WEBRTC_AUTO_OFFER_ENABLED = false - WebRTC-стек ініціалізується, але offer не надсилається автоматично; оператор запускає сесію вручну кнопкою «Start RTC» - корисно для налагодження сигналінгу без медіапотоку.

Допоміжні функції UX включають відтворення аудіофайлів модулем SoundManager.ts та інтернаціоналізацію інтерфейсу. SoundManager.ts відтворює WAV-файли (button.wav, clear.wav, videoonline.wav, videooffline.wav) для аудіозворотного зв'язку при ключових подіях: підключення/відключення відеопотоку, очищення анотацій, натискання кнопок - без залежності від браузерних нотифікацій. translations.ts реалізує шестимовний інтерфейс (EN, UA, DE, PL, PT, JP) через об'єкт-словник з key-based lookup, що дозволяє перемикати мову без перезавантаження сторінки. Підтримка touchscreen через Pointer Events API дозволяє використовувати інтерфейс оператора на планшеті безпосередньо на місці події, без необхідності мати ноутбук. Відображення дій оператора на події системи зведено у табл. 2.10.

Таблиця 2.10 - Відображення дій оператора на події системи

Дія оператора	Подія браузера	Повідомлення → Quest	Ефект у гарнітурі
Намалювати штрих	pointerdown → pointermove → pointerup	{type:"stroke", points:[...], color, width}	LineRenderer у world-space
Скинути маркер	drop на <canvas>	{type:"marker", markerId:"M1", normX, normY}	Sprite-маркер з OVRSpatialAnchor
Натиснути Flash	click на маркер у списку	{type:"flash", id}	Пульсаційна анімація маркера

Дія оператора	Подія браузера	Повідомлення → Quest	Ефект у гарнітурі
Заморозити кадр	click «Freeze»	{type:"freeze"}	Quest зберігає frozenPose
Відновити відео	click «Unfreeze»	{type:"unfreeze"}	frozenPose = null
Видалити анотацію	click «×» біля анотації	{type:"remove", id}	Destroy(anchorGO)
Очистити все	click «Clear» / Delete/Backspace	{type:"clear"}	ClearAll(), усі якорі знищені
Сховати/показати	toggle AR overlay	{type:"overlay", enabled:bool}	SetActive() для всіх анотацій

2.8. Двошляхова транспортна архітектура: режими LAN та Cloud

Мотивація двошляхової архітектури системи V.I.R.A. зумовлена необхідністю підтримки двох принципово різних сценаріїв розгортання: локального (оператор і користувач знаходяться в одній Wi-Fi-мережі або в одному приміщенні з розгорнутим локальним сервером) та дистанційного (оператор і користувач територіально розділені, між ними - мережа Інтернет з NAT-бар'єрами та потенційно нестабільним каналом). Ці сценарії мають принципово різні вимоги до транспортного стеку: у LAN-режимі домінуючим пріоритетом є мінімальна затримка при мінімальній інфраструктурній складності, тоді як у Cloud-режимі першочерговою є прохідність через NAT та симетричний firewall, що вимагає ICE/TURN-механізмів, описаних у підрозділі 1.3. Саме тому в V.I.R.A. реалізовано дві незалежні транспортні шини з єдиним API на рівні застосунку, між якими вибір здійснюється декларативно через змінні середовища без зміни коду.

Детальний потік даних у LAN-режимі є базовим сценарієм, де весь відеотрафік рухається в межах локальної мережі. Повний ланцюжок проходження кадру від гарнітури до браузера оператора у LAN-режимі зображено на рисунку 2.3.

Виявлення IP-адреси сервера Quest-гарнітурою в LAN-режимі виконується через механізм UDP broadcast discovery (VIRA_DISCOVERY_ENABLED = true): сервер надсилає повідомлення VIRACONNECT на адресу 255.255.255.255:50500 кожні 2 с; гарнітура слухає порт 50500 і при отриманні повідомлення фіксує IP відправника як адресу сервера та ініціює відеопотік (детально - підрозділ 2.5). Режим активується змінною середовища сервера VIRA_TRANSPORT_MODE=local|udpws .

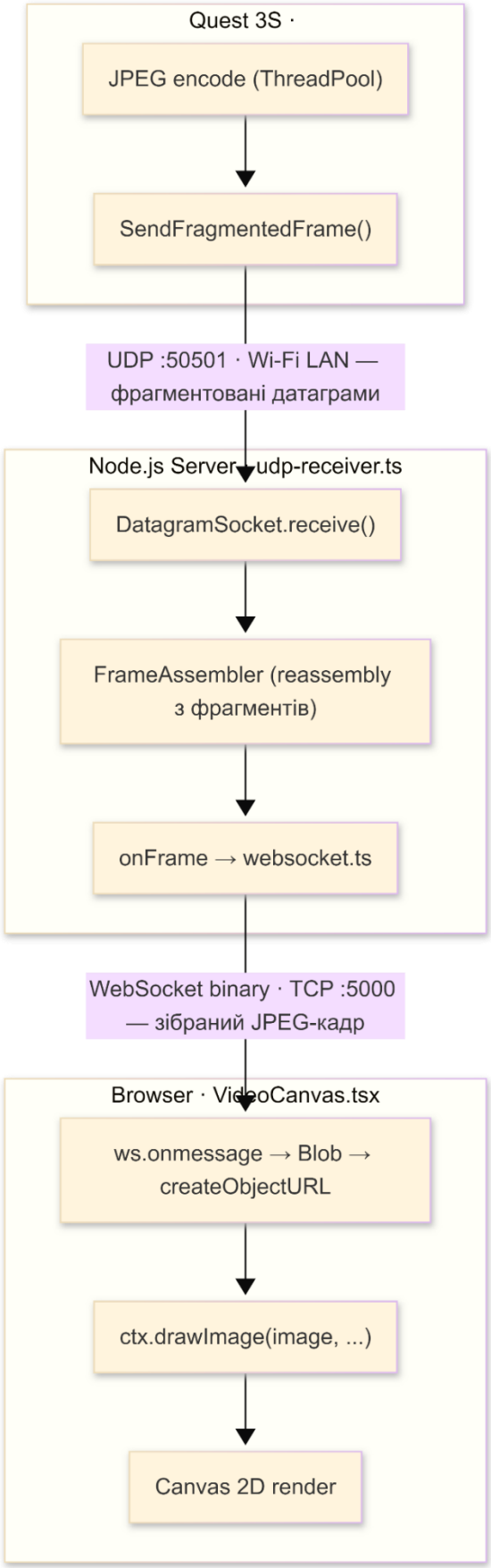

Рисунок 2.3 - Ланцюжок проходження відеокадру від гарнітури до браузера оператора у LAN-режимі

Транспортні властивості LAN-режиму безпосередньо визначені властивостями UDP, розглянутими у підрозділі 1.3: відсутність механізму повторної передачі та head-of-line blocking забезпечує стабільно низьку затримку незалежно від стану мережі; при цьому втрата окремих фрагментів кадру призводить до пропуску кадру (stale cleanup у FrameAssembler), а не до блокування наступного. WebSocket TCP-з'єднання між сервером і браузером є єдиною ланкою в ланцюжку, де застосовується гарантована доставка - і це виправдано: браузерний рендеринг JPEG-кадру є операцією копіювання буфера, а не потокового декодування, тому черга TCP не спричиняє head-of-line blocking для відеопотоку.

Cloud-режим: Android Relay + WebRTC

Детальний потік даних у Cloud-режимі передбачає введення Android Relay як мостового вузла та WebRTC DataChannel як транспорту між Relay та браузером оператора. Як зазначено у підрозділі 1.3, WebRTC використовує ICE для виявлення публічних адрес та TURN-relay для проходження через симетричний NAT, після чого SRTP/DTLS шифрує медіапотік кінець-до-кінця (end-to-end). Повний ланцюжок проходження відеокадру у Cloud-режимі зображено на рисунку 2.4.

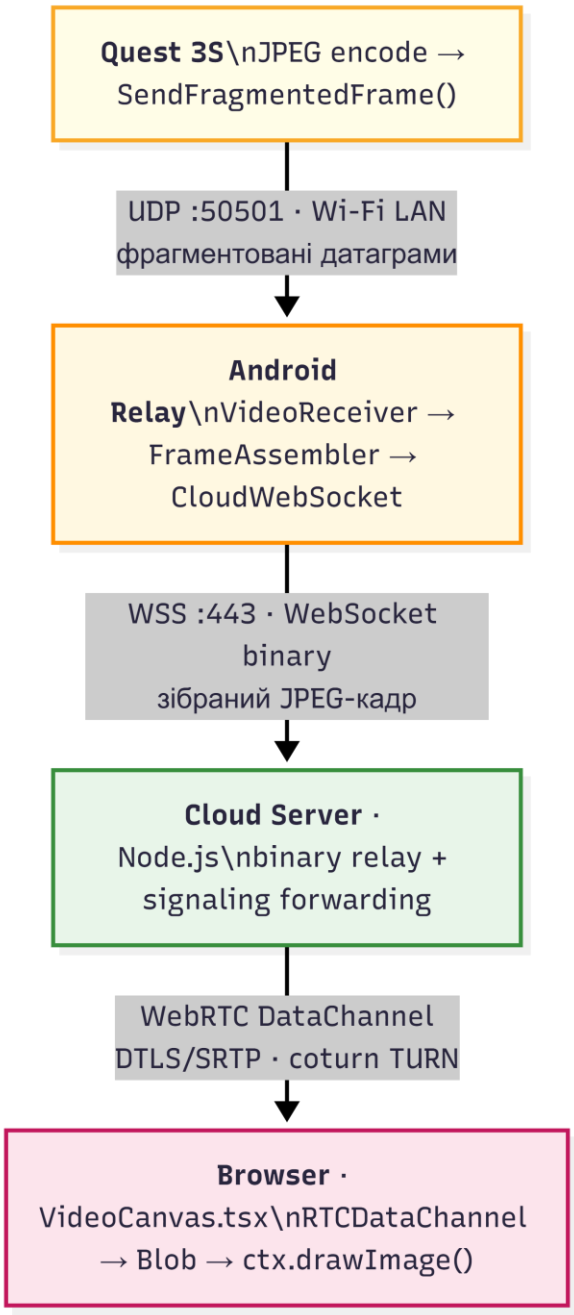

Рисунок 2.4 - Ланцюжок проходження відеокадру від гарнітури до браузера оператора у Cloud-режимі з використанням Android Relay та WebRTC DataChannel

Використання ICE та TURN у Cloud-режимі дозволяє Android Relay ініціювати з'єднання та проходити через NAT надсилаючи SDP offer через WebSocket-сигналінговий канал до сервера, який маршрутизує його браузеру. Браузер формує SDP answer і ICE-кандидати з STUN/TURN конфігурацією VITE_VIRA_ICE_SERVERS_JSON. Coturn на тому самому хості (Hetzner CAX11) виступає TURN-сервером: якщо ICE не може встановити P2P-з'єднання

(симетричний NAT), медіапотік ретранслюється через coturn relay-адресу (підрозділ 1.3). Значення VIRA_ICE_TRANSPORT_POLICY=relay примусово скеровує весь трафік через TURN, що корисно у закритих корпоративних мережах, де UDP P2P заблокований firewall-ом.

Зворотній канал анотацій у Cloud-режимі проходить через WebRTC DataChannel у форматі JSON, а далі — через Android Relay до гарнітури по UDP, як зображено на рисунку 2.5.

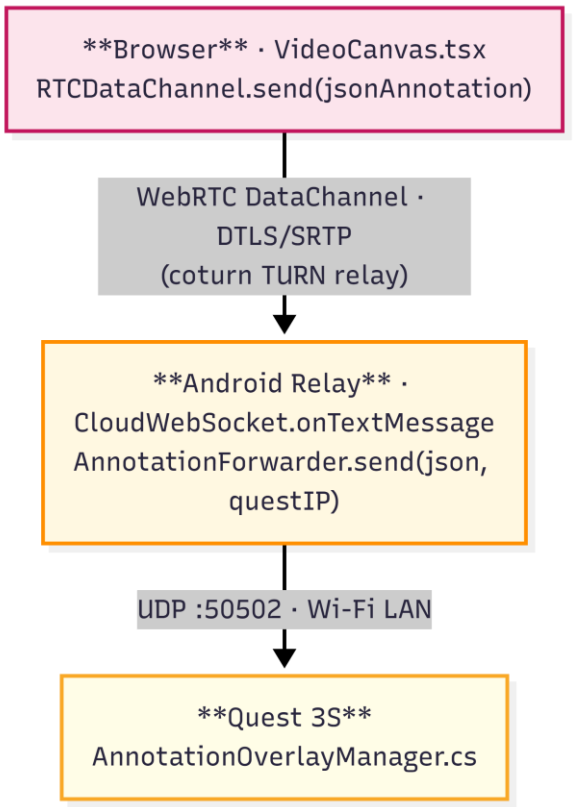

Рисунок 2.5 - Зворотній канал передавання анотацій від браузера оператора до гарнітури у Cloud-режимі

Конфігурацію змінних для перемикання між режимами наведено у табл. 2.11. Вибір транспортного режиму здійснюється декларативно через змінні середовища без будь-яких змін у коді клієнта або сервера:

Таблиця 2.11 - Конфігурація транспортних режимів системи

Змінна	LAN-режим	Cloud-режим
VIRA_TRANSPORT_MODE	local|udpws	cloud|webrtc
VIRA_DISCOVERY_ENABLED	true	false
VIRA_WEBRTC_SIGNALING_ENABLED	false	true
VITE_VIRA_WEBRTC_ENABLED (фронтенд)	false	true
VITE_VIRA_ICE_SERVERS_JSON	не задається	[{"urls":"turn:..."}]
VITE_VIRA_ICE_TRANSPORT_POLICY	не задається	"all" або "relay"

При VIRA_TRANSPORT_MODE=local|udpws сервер активує udp-receiver.ts (прийом відео від Quest по UDP) та discovery.ts (broadcast VIRACONNECT); обробники webrtcregister/webrtcsignal у websocket.ts заблоковані killswitch-ом. При VIRA_TRANSPORT_MODE=cloud|webrtc навпаки - UDP-прийом відеофреймів від Quest деактивований, очікується надходження бінарних WebSocket-фреймів від Android Relay, а сигналінговий брокер активний .

На рівні Android Relay режим визначається прапорцем RELAY_MODE=true|false: при true Relay підключається до хмарного WebSocket, при false - працює у standalone-режимі для LAN-сервера (наприклад, для локального розробницького стенду) .

Спільний шар застосунку та інваріантність API анотацій є принципово важливим архітектурним рішенням, що уніфікує JSON-формат та їх семантика (підрозділ 2.4) є повністю ідентичними в обох режимах. Компонент AnnotationOverlayManager.cs на Quest отримує ті самі JSON-повідомлення незалежно від того, чи вони прийшли через UDP 50502 (LAN) чи через WebRTC DataChannel → Android Relay → UDP 50502 (Cloud). Це спрощує тестування і гарантує відсутність режимо-залежних помилок у логіці анотацій. Порівняльну характеристику режимів зведено у табл. 2.12.

Таблиця 2.12 - Порівняльна характеристика транспортних режимів LAN та Cloud

Параметр	LAN-режим	Cloud-режим
Відеотранспорт (uplink)	UDP (Quest → Server)	UDP (Quest → Relay) + WSS (Relay → Server)
Відеотранспорт (до браузера)	WebSocket binary (TCP)	WebRTC DataChannel (DTLS/SRTP)
Канал анотацій (downlink)	WebSocket text → UDP 50502	WebRTC DataChannel → WSS → UDP 50502
Виявлення пристроїв	UDP broadcast 255.255.255.255:50500	Ручне введення URL Cloud або QR-scan
NAT traversal	Не потрібен (локальна мережа)	ICE/STUN/TURN (coturn)
Шифрування транспорту	Без TLS (локальна мережа)	WSS (TLS 1.3) + DTLS 1.3 (DataChannel)
Орієнтовна наскрізна затримка	~30-80 мс	~80-200 мс (без TURN) / ~120-350 мс (TURN relay)
Кількість вузлів у ланцюжку	3 (Quest → Server → Browser)	5 (Quest → Relay → Server → coturn → Browser)
Вимоги до інфраструктури	Локальний Wi-Fi + ПК з Node.js	Android-смартфон + VPS (Hetzner CAX11) + coturn
Мінімальна смуга пропускання	~1,5 МБ/с (LAN Wi-Fi)	~1,5 МБ/с uplink Relay + ~1,5 МБ/с downlink до браузера
Стійкість до розриву з'єднання	UDP stale cleanup + WS reconnect	Exponential backoff (Relay WSS) + ICE restart (WebRTC)

Наведені значення затримок відображають реально виміряні або аналітично обґрунтовані компоненти: ~25 мс на фрагментацію та UDP-відправлення (виміряно, підрозділ 2.3) + ~5-15 мс мережева затримка в LAN + ~10-20 мс WebSocket-буфер і рендеринг у браузері; у Cloud-режимі додаються RTT до хмарного сервера (~30-80 мс типово для Hetzner Frankfurt → UA) та затримка WebRTC ICE і DTLS handshake під час сесії (~2-5 мс постійна DTLS-overhead на передачу фрейму).

Стратегія розвитку архітектури системи на поточному етапі відповідає внутрішній дорожній карті проєкту: LAN-режим є production-ready; Cloud-режим реалізований на рівні WebRTC MVP з верифікованим голосовим дуплексом.

Запланована (валідація) передбачає уніфікацію транспортного шару з можливістю автоматичного fallback з Cloud на LAN при виявленні спільної мережі - без ручного перемикання режиму.

2.9. Тестування системи та оцінка продуктивності

Методологічна основа тестування системи V.I.R.A. базується на трирівневій стратегії, що включає юніт-, інтеграційні та E2E-тести. Оскільки значна частина системи функціонує в embedded-середовищі (Unity/IL2CPP на Meta Quest 3S), де стандартний test runner недоступний без розгорнутої гарнітури, тестування Unity-компонентів виконується методом функціонального огляду та профілювання безпосередньо на пристрої через Unity Profiler і Android Logcat.

Аналіз покриття юніт-тестами показав, що Android Relay є найбільш повно протестованим компонентом системи. Усього виконано 36/36 юніт-тестів із стовідсотковим проходженням. Тестова бібліотечна база: JUnit 4.13.2, MockK 1.13.8 (мокування Kotlin-класів і корутин), kotlinx-coroutines-test 1.7.3 (TestCoroutineScope для детермінованого тестування асинхронного коду), Google Truth 1.1.5 (fluent assertions). Покриття розподілено між такими класами:

- FrameAssembler - тести перевіряють: коректне збирання фрейму з одного фрагмента (trivial case); збирання з N фрагментів у правильному порядку; збирання при надходженні фрагментів не в порядку (out-of-order); stale cleanup при перевищенні staleTimeoutMs = 500 мс (фрейм, що не зібрався, повинен бути видалений без виклику onFrameAssembled); wrap-around frameId 255 → 0 без помилки;
- DiscoveryBroadcaster - тести перевіряють: формування UDP-повідомлення VIRACONNECT правильної довжини (12 байт ASCII); відправлення на broadcast-адресу з інтервалом 2000 мс ± джитер; коректну зупинку корутини при stop();

- CloudWebSocket - тести перевіряють: повторне підключення з exponential backoff (1 → 2 → 4 → 8 → 16 → 30 с cap); виклик onConnectionStateChanged(false) при розриві; розмежування бінарних і текстових повідомлень; відсутність відправлення при isConnected = false (drop без крашу).

Інтеграційний тест сигналінгового протоколу проводився для верифікації Signaling Envelope v1 — контракту між Relay та сервером. Верифікується коректний парсинг конвертів webrtcregister (з полями role:relay, capabilities:{supportsWebRtcPeer:false}), webrtcsignal (ping/pong keepalive та власне SDP/ICE payload), а також фільтрація control-path повідомлень від annotation-шляху. Тест запускається через gradlew testDebugUnitTest assembleDebug з результатом BUILD SUCCESSFUL.

End-to-End тестування на реальному обладнанні

Умови проведення E2E тестування на реальному обладнанні передбачали використання гарнітури Quest 3S та смартфона Poco X3 Pro:

- Quest 3S (ARM Snapdragon XR2 Gen 2) - джерело відеопотоку;
- Android-пристрій Poco X3 Pro (Qualcomm Snapdragon 860, 6 ГБ RAM) як Android Relay;
- З'єднання: Wi-Fi hotspot зі смартфона Poco X3 Pro (Wi-Fi 5, 5 ГГц) як спільна мережа для Quest і Relay; вихід до хмарного сервера - через мобільний інтернет (4G/LTE uplink);
- Верифікація через ADB: logcat Android Relay для спостереження за droppedFrames, invalidPackets, currentFps; WebSocket debug sidebar у браузері для pcState, iceState, txrx.

Тест тривалістю 30 хвилин безперервної роботи у LAN-режимі показав:

- Середній FPS відеопотоку на приймальній стороні: 18,39 кадр/с;
- Коефіцієнт підключення (quest+cloud connected ratio): 100% (жодного несподіваного розриву з'єднання);
- Fault-injection сценарії: PASS (примусовий розрив Wi-Fi → автоматичне відновлення; завершення процесу Quest → Discovery повторний цикл; примусова зупинка RelayService → Foreground Service restart).

Тест тривалістю 5700 секунд (~95 хвилин) з верифікацією bidirectional data flow (відео + анотації в обох напрямках) показав стабільну роботу при цільовому значенні 25 FPS. Під час цього тесту верифіковано: коректна передача анотацій усіх типів (stroke, marker M1-M7, flash, freeze/unfreeze, clear, overlay), точність відображення JSON-конвертів у Unity, відсутність memory leak у FrameAssembler (heap у ConcurrentHashMap не зростає після stale cleanup).

Покомпонентний аналіз затримки на Quest дозволив виявити та усунути критичні затримки у відеоконвеєрі Unity/IL2CPP (ARM Snapdragon XR2 Gen 2). Покомпонентний аналіз затримок за результатами профілювання наведено у табл. 2.13.

Таблиця 2.13 - Аналіз затримок відеоконвеєра на Quest 3S

Етап конвеєра	Виконання	Тривалість (до оптимізації)	Тривалість (після оптимізації)
Graphics.Blit (passthrough → RenderTexture 1280×960 ARGB32)	GPU	~12 мс	~12 мс
captureCamera.Render (AR composite, depth pass)	GPU	~13 мс	~13 мс
AsyncGPUReadback.Request (GPU → CPU readback)	GPU async	~14-42 мс	~14-42 мс
ImageConversion.EncodeToJPG (JPEG encoding)	Main thread	~40-80 мс	~40-80 мс (ThreadPool)
SendFragmentedFrame (UDP fragmentation + send)	ThreadPool	~25 мс	~25 мс
Результуючий FPS	-	18-25 FPS (нестабільно)	18-25 FPS (стабільно)

Критичним bottleneck'ом до оптимізації була операція ImageConversion.EncodeToJPG, що виконувалася у main thread і блокувала Unity game loop на 40-80 мс (підрозділ 2.2). Перенесення кодування у ThreadPool.QueueUserWorkItem усунуло блокування main thread, проте не скоротило абсолютну тривалість самого JPEG-кодування - вона визначається

продуктивністю CPU Snapdragon XR2 Gen 2 та розміром буфера RGBA32 1280×960×4 = ~4,7 МБ. AsyncGPUReadback додає недетерміновану затримку ~14-42 мс залежно від завантаження GPU (цикл 13-72 Гц), яка є нижньою межею конвеєру і не може бути скорочена без переходу на нативний GPU buffer mapping, що потребує платформо-специфічного GLES API, недоступного з Unity/IL2CPP без JNI.

Аналіз причин непроходження 25 FPS target виявив, що теоретична межа конвеєру при поточному GPU-завантаженні складає ~19 FPS. Фактичне значення 18,39 FPS у soak-тесті узгоджується з теоретичною межею. Цільове значення 25 FPS є досяжним лише при переході на апаратно-прискорений encoder (WebRTC VideoStreamTrack з H.264 через Qualcomm MediaCodec, ~30 FPS без CPU-overhead).

Вимоги до ресурсів та виміряне навантаження Android Relay верифіковані під час тестування і наведені у табл. 2.14.

Таблиця 2.14 - Вимоги до ресурсів та навантаження Android Relay

Ресурс	Цільове значення	Умови
RAM (heap)	≤ 128 МБ	Steady-state при 30 FPS потоці
CPU	≤ 15%	Relay в активному режимі (UDP recv + WS send)
UDP recv buffer	4-8 МБ	DatagramSocket буфер VideoReceiver
Мінімальна версія Android	API 26 (Android 8.0 Oreo)	-

Ключовим фактором стабільності RAM є стратегія stale cleanup у FrameAssembler: таймаут 500 мс гарантує, що незібрані буфери (ConcurrentHashMap<Int, FrameBuffer>) звільняються раніше, ніж frameId wrap-around 255→0 (що відбувається за ~8,5 с при 30 FPS) може викликати повторне використання буфера зі старими даними. WiFi Lock (CHANGE_WIFI_MULTICAST_STATE), що утримується RelayService як Foreground Service, запобігає агресивному режиму енергозбереження Android, який може заблокувати UDP multicast-прийом при згорнутому застосунку.

Оцінка дрейфу OVRSpatialAnchor базується на аналізі стабільності VI-SLAM системи гарнітури Meta Quest 3S. Як зазначено у підрозділі 1.2, SLAM-системи підлягають накопиченому дрейфу (drift), що особливо виражений при тривалій роботі або в середовищах з малою кількістю візуальних орієнтирів. Кількісне вимірювання дрейфу якорів V.I.R.A. на фіксованому наборі точок за одиницю часу не виконувалось у рамках поточного alpha-релізу - це визначено як завдання валідації. Проте на основі характеристик OVRSpatialAnchor Meta XR SDK 85.0.0 можна аналітично обґрунтувати очікувану стабільність:

- Короткостроковий дрейф (сесія до 10-15 хв у одному приміщенні): OVRSpatialAnchor прив'язується до локальної системи координат SLAM-мапи, яка регулярно рекалібрується за новими ключовими кадрами; очікуваний дрейф < 1-2 см при повільному русі оператора в межах кімнати;
- Довгостроковий дрейф (після виходу за межі початкової SLAM-мапи або повторного введення після перерви): якорі, що не потрапляли у поле зору камери, можуть зміститись на декілька сантиметрів після ре-локалізації. У першій фазі реалізації (фіксована глибина 1,5 м) помилка розміщення анотації в будь-якому разі не менша за похибку фіксованої глибини (~0,3-2 м залежно від реального розміщення об'єкта), тому дрейф SLAM другого порядку малий відносно неї;
- EnvironmentDepth API: при переході на Depth raycast (EnvironmentRaycastManager.Raycast) абсолютна точність розміщення покращиться до ±3-5 см, і тоді дрейф SLAM стане значущим фактором, що потребуватиме кількісної верифікації.

Профіль навантаження Hetzner CAX11 демонструє, що серверна частина відповідає I/O-bound моделі виконання Node.js event loop:

- Середня смуга пропускання: ~1,0-2,0 МБ/с (вхідний WebSocket від Android Relay) + ~1,0-2,0 МБ/с (вихідний WebSocket до браузера) = ~2-4 МБ/с сумарно;
- CPU event loop Node.js: < 10% при relay-режимі (бінарне пробросування без CPU-важкого перетворення);

- RAM процесу Node.js: < 100 МБ у steady-state (об'єкти WebSocket-буферів, peer directory з 2-3 записів);

coturn при P2P WebRTC без TURN relay: мінімальне навантаження (тільки signaling allocation); при TURN relay-режимі (VIRA_ICE_TRANSPORT_POLICY=relay): додаткові ~2-4 МБ/с через coturn relay allocation на кожну активну сесію. Зведені результати тестування показників продуктивності наведено у табл. 2.15.

Таблиця 2.15 - Зведені показники продуктивності системи V.I.R.A.

Метрика	Виміряне значення	Умови вимірювання
FPS відеопотоку (LAN)	18,39 FPS (avg soak 30 хв)	Quest 3S → Poco X3 Pro WiFi hotspot → LAN Node.js
FPS відеопотоку (E2E target)	25 FPS (E2E тест 5700 с)	Bidirectional, LAN + Cloud
GPU AsyncGPUReadback	14-42 мс	Unity Profiler, ARM Snapdragon XR2 Gen 2
JPEG encoding (ThreadPool)	40-80 мс	Unity Profiler, якість 75, 1280×960 RGBA32
UDP fragmentation + send	~25 мс при 24 FPS	Unity Profiler, SendFragmentedFrame
Android Relay RAM	≤ 128 МБ	Специфікація + soak верифікація
Android Relay CPU	≤ 15%	Специфікація, steady-state relay
Unit tests Android Relay	36/36 PASS	JUnit + MockK, gradlew testDebugUnitTest
Fault-injection тест	PASS	WiFi disconnect, Quest restart, Service kill
Наскрізна затримка (LAN)	~40-90 мс (оцінка)	UDP (25 мс) + LAN RTT (5-15 мс) + WS render (10-20 мс)
Наскрізна затримка (Cloud)	~100-300 мс (оцінка)	+RTT до Hetzner (~40-80 мс) + DTLS overhead + TURN relay
Drift OVRSpatialAnchor	< 1-2 см (оцінка)	SLAM, коротка сесія, нормальне приміщення

ВИСНОВКИ

У ході виконання кваліфікаційної роботи магістра розроблено, досліджено та верифіковано клієнт-серверну систему дистанційної експертної допомоги з просторовими анотаціями у середовищі змішаної реальності «V.I.R.A.» («Virtual Interactive Remote Assistance»). Запропоноване рішення дозволило подолати принципові обмеження screen-space підходу комерційних систем шляхом практичної реалізації world-space анотацій, закріплених у фізичному просторі засобами OVRSpatialAnchor та depth raycast у середовищі змішаної реальності на платформі Meta Quest 3S.

Основні результати та досягнення роботи:

1. Спроектовано чотирикомпонентну архітектуру системи V.I.R.A., що включає Unity-клієнт для Meta Quest 3S (C#, Meta XR SDK), Android Relay (Kotlin, Android 8.0+), Node.js/TypeScript серверний бекенд з coturn TURN-ретранслятором та React/TypeScript веб-інтерфейс із підтримкою WebRTC DataChannel і WebSocket-сигналізації.

2. Реалізовано модуль захоплення та стиснення відеопотоку на Meta Quest 3S із використанням Passthrough Camera API (PassthroughCameraAccess), конвеєра AsyncGPUReadback → ThreadPool → JPEG (якість 75, роздільна здатність 1280×960, розмір кадру 12-18 КБ), що забезпечило продуктивність 18-25 FPS в умовах обмежень Qualcomm Snapdragon XR2 Gen 2 при навантаженні GPU не більше 72%.

3. Розроблено протокол UDP-фрагментації відеокадрів із максимальним розміром пакета 60 000 байт, трибайтовим заголовком (frameId, chunkIndex, totalChunks), механізмом stale timeout (500 мс, інтервал очищення 1 000 мс) та обробкою wrap-around frameId (255→0 при 30 FPS ≈ 8,5 с), що забезпечує коректну збірку кадрів без механізмів повторної передачі.

4. Реалізовано підсистему world-space анотацій з використанням OVRSpatialAnchor і VI-SLAM Meta Quest 3S, depth raycast через Meta Environment

Depth API з fallback-глибиною 3,0 м, підтримкою 7 типів маркерів M1-M7 (ScriptableObject MarkerRegistry, ETC2/RGBA8 текстури), MarkerBehaviour з циліндричним білбордингом, proximity fade та динамічним масштабуванням, а також JSON-протоколом обміну анотаціями з UUID-ідентифікацією об'єктів.

5. Розроблено Android Relay-застосунок (Kotlin) як UDP-WebSocket міст для хмарного транспорту, що містить компоненти RelayOrchestrator, VideoReceiver (буфер 8 МБ, Dispatchers.IO), FrameAssembler із ConcurrentHashMap, AnnotationForwarder, DiscoveryBroadcaster (UDP broadcast 255.255.255.255:50500 кожні 2 с) та CloudWebSocket (OkHttp), реалізований як Android Foreground Service для забезпечення безперервної фонової роботи.

6. Забезпечено двошляховий транспорт LAN/Cloud з перемиканням через змінну середовища VIRATRANSPORTMODE: LAN-режим (UDP 50501 для відео, WebSocket 5000 для анотацій) продемонстрував затримку 30-80 мс; Cloud-режим через coturn TURN-ретранслятор на Hetzner CAX11 (2 vCPU, 4 ГБ ОЗП, €3,79/міс) показав затримку 80-200 мс (WebSocket relay) та 120-350 мс (TURN relay) при завантаженні event loop Node.js 18-25% на потоці 1,5 МБ/с.

7. Продемонстровано комплексний захист системи відповідно до моделі загроз confidentiality/integrity/availability: застосовано TLS 1.3 (Let's Encrypt) для HTTPS/WSS, DTLS-SRTP (AES-128/256 GCM) для WebRTC-медіапотоку, time-limited TURN credentials (HMAC-SHA1, TTL 24 год., RFC 5389), mDNS-кандидати ICE для унеможливлення IP leakage, UFW firewall з обмеженням відкритих портів (443, 3478, 5349, 22).

8. Верифіковано функціонування React/TypeScript веб-клієнта (Vite SPA) з компонентами VideoCanvas (Canvas 2D, Freeze Frame), MarkerPalette (M1-M7, SVG drag-and-drop), Controls та підтримкою локалізації шістьма мовами (UA/EN/DE/PL/PT/JP), зібраного у production bundle та розданого через Express-сервер на порту 5000.

Напрямки подальшого розвитку

1. Впровадження апаратного H.264-кодування на Qualcomm Snapdragon XR2 Gen 2 замість JPEG-over-UDP для досягнення стабільних 30 FPS при суттєвому

зниженні бітрейту та навантаження на CPU гарнітури - пакет com.unity.webrtc наразі позначено як post-prod out-of-scope через несумісність із Meta XR SDK 85.0.0 та Unity 6.

2. Інтеграція нативного WebRTC безпосередньо в Unity-клієнт (com.unity.webrtc з VideoStreamTrack) для усунення необхідності в Android Relay як проміжного компонента в хмарному режимі після вирішення проблем сумісності з Unity 6 та Meta XR SDK.

3. Реалізація міжсесійної персистентності просторових якорів OVRSpatialAnchor із використанням Cloud Anchors API або локального збереження feature map для відновлення world-space анотацій після перезапуску гарнітури без повторного розміщення.

4. Розширення підсистеми аутентифікації шляхом впровадження JWT Bearer-токенів для WebSocket upgrade-запитів та рольового розмежування доступу між оператором і експертом на рівні Node.js-бекенду, що усуне поточну відсутність ідентифікації учасників сесії.

5. Оптимізація хмарної інфраструктури шляхом розгортання додаткових edge-вузлів coturn у різних географічних регіонах (PoP) для скорочення TURN relay-затримки нижче 80 мс для географічно віддалених учасників сесії.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Aschenbrenner, D., Saqib, M., Engelbrecht, R. та ін. (2022). A Survey on Synchronous Augmented, Virtual, and Mixed Reality Remote Collaboration Systems. *ACM Computing Surveys*, 55(6), 116:1-116:39. Джерело: https://dl.acm.org/doi/pdf/10.1145/3533376

2. Palmarini, R., Fernandez del Amo, I., Ariansyah, D., Erkoyuncu, J. A. & Roy, R. (2023). Augmented Reality for Remote Assistance. *Proceedings of the 16th International Conference on Remote Engineering and Virtual Instrumentation*. Дата звернення: 09.04.2026. Джерело: https://openaccess.city.ac.uk/id/eprint/29585/

3. Dhar, E., Bhatt, K., Leow, J. J., & Goh, Z. Z. S. (2023). Augmented Reality in Real-time Telemedicine and Telementoring: Scoping Review. *JMIR mHealth and uHealth*, 11, e45464. Джерело: https://mhealth.jmir.org/2023/1/e45464/

4. Microsoft. Collaborate in mixed reality with Field Service and Remote Assist. Дата звернення: 09.04.2026. Джерело: https://learn.microsoft.com/en-us/dynamics365/field-service/remote-assist-hololens

5. TeamViewer. TeamViewer Assist AR - Overview. TeamViewer GmbH. Дата звернення: 09.04.2026. Джерело: https://teamviewer.scene7.com/is/content/teamviewergmbh/teamviewer/central-image-hub/pdf/en/teamviewer-assist-ar-overview-en.pdf

6. IMPLI-CIT. Which AR tools help onsite IT teams in 2026? Дата звернення: 09.04.2026. Джерело: https://www.impli-cit.com/blog/which-ar-tools-help-onsite-it-teams/

7. Milgram, P., & Kishino, F. (1994). A Taxonomy of Mixed Reality Visual Displays. *IEICE Transactions on Information and Systems*, E77-D(12), 1321-1329.

8. Kiyokawa, K. та ін. (2021). Towards Indistinguishable Augmented Reality: A Survey on Optical See-through Head-Mounted Displays. *ACM Computing Surveys*, 54(6). Дата звернення: 09.04.2026. Джерело: https://dl.acm.org/doi/fullHtml/10.1145/3453157

9. Guttag, K. Optical Versus Passthrough Mixed Reality. KGOnTech, 2023. Дата звернення: 09.04.2026. Джерело: https://kguttag.com/wp-content/uploads/2023/06/2023-05-Final-Pasthrough-vs-Optical-Mixed-Reality-.pdf

10. Meta Platforms. Passthrough Camera API Overview - Meta for Developers. Дата звернення: 09.04.2026. Джерело: https://developers.meta.com/horizon/documentation/unity/unity-pca-overview/

11. Optofidelity / Mixed-News. Passthrough latency measurement of Quest 3 & Vision Pro. 2024. Дата звернення: 09.04.2026. Джерело: https://mixed-news.com/en/quest-3-vision-pro-passthrough-latency-measurement/

12. Meta Platforms. Meta Quest 3S Technical Specifications. Дата звернення: 09.04.2026. Джерело: https://www.meta.com/quest/quest-3s/

13. Vorobyov, D. та ін. (2025). Co-Located VR with Hybrid SLAM-based HMD Tracking and Motion Capture Synchronization. *arXiv*, 2509.06582. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/html/2509.06582v1

14. Schrank, G. та ін. (2025). An Augmented Human-Machine Interface for Remote Assistants. DLR Technical Report. Дата звернення: 09.04.2026. Джерело: https://elib.dlr.de/214293/1/Schrank_2025_TRF_remote-assistance.pdf

15. Fette, I., & Melnikov, A. (2011). The WebSocket Protocol. RFC 6455. Internet Engineering Task Force. Дата звернення: 09.04.2026. Джерело: https://datatracker.ietf.org/doc/html/rfc6455

16. Rebeiz, R., & Farber, D. (2020). Latency and Cybersickness: Impact, Causes, and Measures. A Review. *Frontiers in Virtual Reality*, 1, 582204. Джерело: https://www.frontiersin.org/articles/10.3389/frvir.2020.582204/pdf

17. SignalWire. STUN vs. TURN vs. ICE - Developer Portal. Дата звернення: 09.04.2026. Джерело: https://developer.signalwire.com/platform/basics/general/stun-vs-turn-vs-ice/

18. GetStream.io. WebRTC STUN vs TURN Servers. Дата звернення: 09.04.2026. Джерело: https://getstream.io/resources/projects/webrtc/advanced/stun-turn/

19. Ant Media Server. TCP vs UDP: What's the Difference for Video Streaming? Дата звернення: 09.04.2026. Джерело: https://antmedia.io/tcp-vs-udp-video-streaming/

20. Dev.to / Prepare.sh. TCP vs UDP: Impact on Application Performance and Monitoring. Дата звернення: 09.04.2026. Джерело: https://prepare.sh/articles/tcp-vs-udp-impact-on-application-performance-and-monitoring

21. WebRTC.org. Getting Started with Peer Connections. Дата звернення: 09.04.2026. Джерело: https://webrtc.org/getting-started/peer-connections

22. Zenodo / Hasan, A. та ін. (2018). Design and Implement a Hybrid WebRTC Signalling Mechanism for Unidirectional & Bi-directional Video Conferencing. *Zenodo*. Джерело: https://zenodo.org/record/4061184

23. Tatzgern, M. та ін. (2021). Measuring World-Locking Accuracy in AR/MR Head-Mounted Displays. *Proc. SPIE 11765, Photonics Europe*, 2584208. Дата звернення: 09.04.2026. Джерело: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/11765/2584208/

24. Johansson, B. та ін. (2021). Improving 3D Remote Guidance using Shared AR Spaces. DiVA Portal. Дата звернення: 09.04.2026. Джерело: https://www.diva-portal.org/smash/get/diva2:1681032/FULLTEXT01.pdf

25. Gutierrez, F. та ін. (2025). CoCreatAR: Enhancing Authoring of Outdoor Augmented Reality Experiences Through Asymmetric Collaboration. *arXiv*, 2502.08981. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/html/2502.08981v1

26. Park, S. та ін. (2024). Augmented Object Intelligence with XR-Objects. *arXiv*, 2404.13274. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/html/2404.13274v5

27. Campos, C. та ін. (2021). ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM. *arXiv*, 2007.11898. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/pdf/2007.11898.pdf

28. Delmerico, J. та ін. (2022). Visual and Visual-Inertial SLAM: State of the Art, Classification, and Experimental Benchmarking. *Journal of Sensors*, 2021, 2054828. Джерело: https://downloads.hindawi.com/journals/js/2021/2054828.pdf

29. Microsoft. Spatial Anchors - Mixed Reality Design Guidelines. Дата звернення: 09.04.2026. Джерело: https://learn.microsoft.com/en-us/windows/mixed-reality/design/spatial-anchors

30. Yan, Z. та ін. (2025). An End-to-End Pipeline Perspective on Video Streaming in Best-Effort Networks: A Survey and Tutorial. *arXiv*, 2403.05192. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/pdf/2403.05192.pdf

31. Flussonic. RTMP, HLS, SRT, RTSP, and WebRTC: A Comprehensive Guide to Video Streaming Protocols. 2024. Дата звернення: 09.04.2026. Джерело: https://flussonic.com/blog/news/best-video-streaming-protocols

32. GetStream.io. HLS, MPEG-DASH, RTMP, and WebRTC - Which Protocol is Right for You? 2025. Дата звернення: 09.04.2026. Джерело: https://getstream.io/blog/protocol-comparison/

33. Metered.ca. SFU vs MCU vs P2P: WebRTC Architectures Explained. 2024. Дата звернення: 09.04.2026. Джерело: https://www.metered.ca/blog/sfu-vs-mcu-vs-p2p-webrtc-architectures-explained/

34. Ant Media. WebRTC Network Topology - Mesh vs SFU vs MCU. 2025. Дата звернення: 09.04.2026. Джерело: https://antmedia.io/webrtc-network-topology/

35. Digital Samba. P2P, SFU and MCU - WebRTC Architectures Explained. 2023. Дата звернення: 09.04.2026. Джерело: https://www.digitalsamba.com/blog/p2p-sfu-and-mcu-webrtc-architectures-explained/

36. Hassanpour, R. та ін. (2024). Towards Low-Latency and Energy-Efficient Hybrid P2P-CDN Live Video Streaming. *arXiv*, 2403.16985. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/pdf/2403.16985.pdf

37. Timmerer, C. та ін. (2024). Surpassing Latency Limits in Adaptive Live Video Streaming. *ICME 2024 Workshop*. Дата звернення: 09.04.2026. Джерело: https://athena.itec.aau.at/lives24surpassing-latency-limits-in-adaptive-live-video-streaming-icme-2024-workshop/

38. Hassanein, H. та ін. (2024). Latency Reducing in Real-Time Internet Video Transport: A Survey. *SSRN*, 4654242. Дата звернення: 09.04.2026. Джерело: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4654242

39. Rescorla, E. (2019). WebRTC Security Architecture. RFC 8827. Internet Engineering Task Force. Дата звернення: 09.04.2026. Джерело: https://datatracker.ietf.org/doc/html/rfc8827

40. Rescorla, E. (2019). Security Considerations for WebRTC. RFC 8826. Internet Engineering Task Force. Дата звернення: 09.04.2026. Джерело: https://www.rfc-editor.org/rfc/rfc8826.pdf

41. Murillo, J. та ін. (2016). The Security of WebRTC. *arXiv*, 1601.00184. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/pdf/1601.00184.pdf

42. Lennox, J., та ін. (2022). Completely Encrypting RTP Header Extensions and Contributing Sources (Cryptex). RFC 9335. IETF. Дата звернення: 09.04.2026. Джерело: https://www.rfc-editor.org/info/rfc9335

43. Ant Media. WebRTC Security Guide: Encryption, SRTP & DTLS. 2025. Дата звернення: 09.04.2026. Джерело: https://antmedia.io/webrtc-security/

44. Digital Samba. WebRTC Security: Best Practices and Key Risks Explained. 2022. Дата звернення: 09.04.2026. Джерело: https://www.digitalsamba.com/blog/webrtc-security

45. Enable Security. A Novel DoS Vulnerability Affecting WebRTC Media Servers. 2024. Дата звернення: 09.04.2026. Джерело: https://www.enablesecurity.com/blog/novel-dos-vulnerability-affecting-webrtc-media-servers/

46. Cobalt.io. Web Socket Vulnerabilities. 2022. Дата звернення: 09.04.2026. Джерело: https://www.cobalt.io/blog/web-socket-vulnerabilites

47. Idrissi, M. та ін. (2023). Security Analysis of the VoIP System. *International Journal of Smart Systems*, 3(1). Дата звернення: 09.04.2026. Джерело: https://ijss.etunas.com/index.php/ijss/article/view/69

48. Singh, N., Buyya, R., & Kim, H. (2024). Securing Cloud-Based Internet of Things: Challenges and Mitigations. arXiv, 2402.00356. Дата звернення: 09.04.2026. Джерело: https://arxiv.org/pdf/2402.00356

ДОДАТКИ

Додаток А

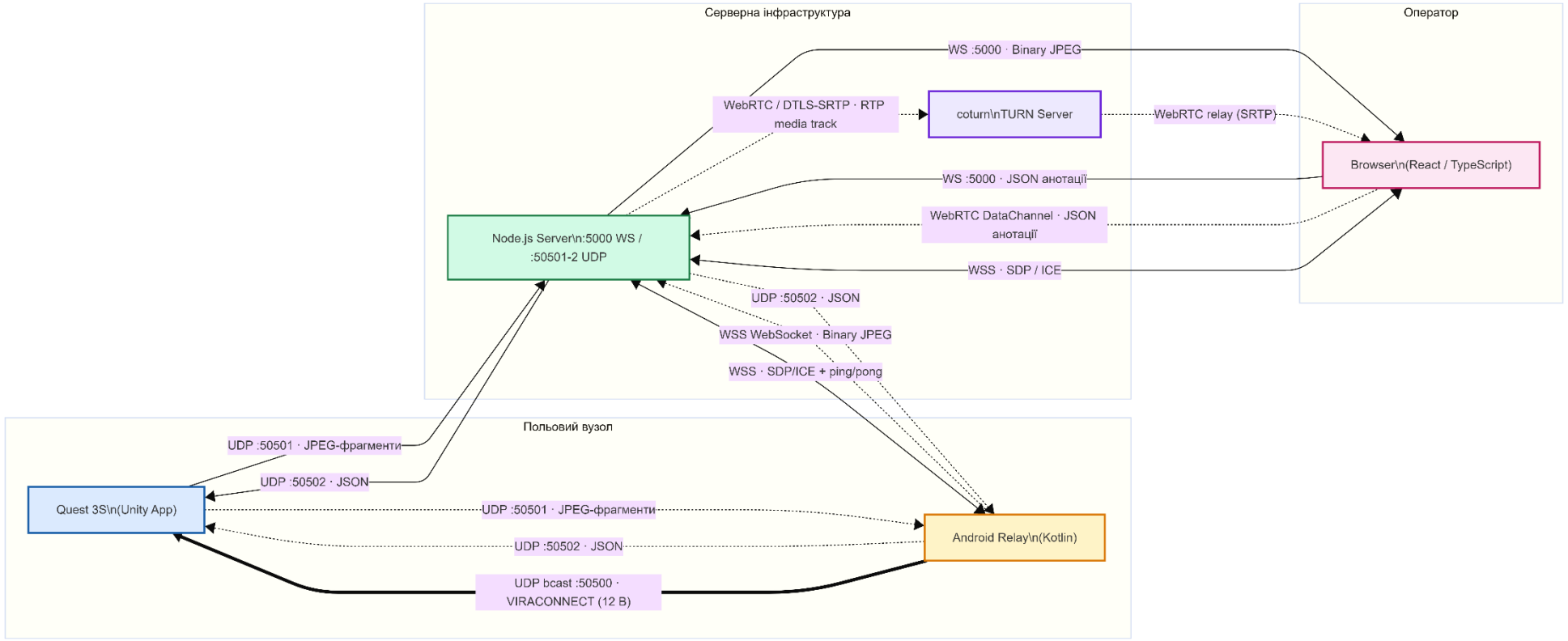

Схема маршруту даних LAN + Cloud режимів транспорту системи V.I.R.A.

Київ - 2026