N1KURA LAB

Архитектура и ключевые модули ProtoChatX

ProtoChatX представляет собой полностью автономный мессенджер с клиентом на одной HTML-странице и серверной частью-реле на Node.js. Основные компоненты и технологии:

- **WebSocket-соединение** – используется постоянное WS-соединение для авторизации и обмена всеми сообщениями и сигналами вызова. Сервер (`relay.js`) поднимает WebSocket-сервер (по умолчанию порт 3001) и обрабатывает входящие пакеты JSON. Реализованы ограничения по размеру (~8 МБ) и частоте сообщений (не более 180 в минуту на соединение) – это защищает от перегрузки и DoS-атак [1] [2]. Некорректный JSON парсится через try/catch, при ошибке клиент получает `error: BAD_JSON` [3]. Таким образом, сервер не выполнит посторонний код, и криво сформированные пакеты просто отвергаются.

- **WebRTC и сигнальный протокол** – для аудио/видео звонков используется WebRTC, а все сигналы (начало вызова, принятие, завершение, обмен SDP/ICE) идут через тот же WebSocket. Сервер выступает только транспортом сигнализации, не смешиваясь в сам поток медиа. Ключевые типы сигналов: `call_invite` (исходящий вызов), `call_accept` (принятие), `call_cancel` / `call_end` / `call_busy` (отмена, завершение, занято) – они *не* помещаются в очередь офлайн и доставляются только онлайн [4]. Обмен SDP-предложениями и ответами (`webrtc_offer`, `webrtc_answer`) и ICE-кандидатами (`webrtc_ice`) также происходит через сообщения WS; эти пакеты теперь **не** считаются “no-queue” (их могут сохранять в офлайн-очереди, но с коротким TTL) [4] [5]. Это сделано для борьбы с «призрачными звонками»: например, `call_invite` хранится не более **45 секунд** (`CALL_INVITE_TTL_MS = 45000`) [6], по истечении удаляется, чтобы пользователь не видел пропущенный звонок спустя долгое время. TURN-креденшлы выдаются по запросу `turn_cred` – сервер сгенерирует логин/пароль по схеме HMAC-SHA1 (если настроен `TURN_SECRET`) сроком на ~1 час [7] [8] и вернёт клиенту вместе с списками TURN/STUN адресов [9]. TURN-сервер (например, coturn) используется для обхода NAT, если прямое P2P недоступно.

- **Очередь сообщений и офлайн-хранилище** – сервер **хранит** несразу доставленные сообщения во встроенной очереди. Каждому пользователю в `queue.json` соответствует массив не доставленных элементов (шифротекстов). При отправке сообщения, если получатель офлайн, сервер возвращает отправителю статус `queued` и сохраняет пакет в файле [10] [11]. Максимум на пользователя – 200 сообщений или ~2 МБ данных, старые сверх лимита или старше 7 дней (`QUEUE_TTL_MS`) автоматически очищаются [12] [5]. Для некоторых **служебных пакетов** включён режим *«no-queue»*, то есть не сохранять офлайн (например, сигналы звонков `call_*`, `ping/pong` присутствия) [4]. Это предотвращает ложные срабатывания: напр. если вызывающий отменил звонок, `call_cancel` не будет воспроизведён позже из очереди, вызывая «призрак». В офлайн-очереди предусмотрена дедупликация: для повторных `call_invite` от одного отправителя старый удаляется, для повторных `webrtc_offer/answer` по данному звонку хранится только последнее (чтобы не накапливать устаревшие SDP) [13] [14]. При логине клиента сервер автоматически **flush-ит очередь**: все накопленные сообщения отправляются клиенту подряд, после чего

удаляются из `queue.json` [15] [16]. Таким образом, истории на сервере не хранится дольше необходимого – только временное буферизование для офлайна.

- **Firebase Push-уведомления** – реализована интеграция с Firebase Cloud Messaging (FCM) для фоновых уведомлений на Android. В клиент встроен Capacitor Push Plugin: после авторизации приложение получает FCM-токен и высылает его серверу (`push_register`) [17] [18]. Сервер сохраняет `fcm_token` в профиле пользователя и при поступлении новых сообщений либо звонков офлайн-адресату шлёт пуш. Контент push минимален: **сообщения** – тип *msg* и ID отправителя, без текста (сервер отсылает notification с заголовком *"Новое сообщение"* без раскрытия содержания [19] [20]), **звонки** – push с типом *call* или *call_ctl* и параметрами (действие, ID звонящего, SID звонка) [21] [22]. Для вызова priority=high и collapseKey `pc_call_<from_id>` (чтобы пуши звонка склеивались). На устройстве эти push обрабатываются: *call* запускает системный экран звонка (звонок «будит» телефон), а *call_ctl* (например, `call_cancel`) сообщает приложению остановить звонок, чтобы рингтон отключился [23] [24]. Такая схема обеспечивает, что при входящем звонке в офлайне телефон зазвонит, но при отмене вызова звон не будет «повисать». Важно, что *пуши не содержат конфиденциальных данных* – только служебный сигнал.

- **Хранилище данных и модули** – вместо СУБД сервер использует простые JSON-файлы: `users.json` (база пользователей), `invites.json` (актуальные инвайт-коды для регистрации) и `queue.json` (очередь сообщений). Доступ к файлам синхронизирован: реализована простая блокировка на основе создания временного lock-файла `.state.lock` [25] [26] и `withStateLock`-функции для атомарных операций (например, одновременная регистрация по одному инвайту в гонке). Этот подход упрощает развёртывание (не требуется отдельная БД), однако требует защитить сами файлы на сервере. **Users.json** содержит: числовой ID, **Secret** (секретный ключ аутентификации), никнейм, флаг `enabled`, публичный E2E-ключ (JWK) и прочие поля (например, пометка, дата создания, токен FCM). **Invites.json** хранит инвайты как хэш-код => объект с параметрами (сколько использований осталось, срок годности, зарезервированный ID и примечание) [27] [28]. **Защита файлов:** при обновлении используются временные копии и rename – во избежание порчи данных (например, пишется `users.json.tmp`, затем заменяется файл) [29] [30]. Все файлы читаются целиком в память для операций – при малом числе пользователей это не проблема, но при росте масштабов стоит учесть, что каждый логин перечитывает весь users.json.

Вывод (Архитектура): Архитектура построена по принципу *клиент – облегчённый сервер-реле*. Сервер **не** хранит и не видит открытый контент сообщений – только пересылает шифрованные блоки, optionally сохраняя их до доставки. Решения, вроде FCM push и TURN, интегрированы корректно, дополняя функциональность (офлайн уведомления, звонки через NAT). Простое файловое хранилище упрощает установку, но накладывает ограничения по масштабируемости (см. ниже). В целом модули (WebSocket, WebRTC, push, offline-очередь) взаимодействуют согласованно. Стоит отметить четкое разделение: файл `relay.js` отвечает за основную логику протокола и данных, а `admin-api.js` – за административные задачи.

Безопасность: шифрование и защита данных

Безопасность в ProtoChatX продумана на нескольких уровнях:

- **Сквозное шифрование (End-to-End, E2EE)** – все личные сообщения шифруются **на клиентах**. Перед отправкой текст или вложение пакуется в JSON с полем `kind` (тип сообщения, напр. *chat*, *avatar*, *call_invite* и т.д.) и необходимыми данными, затем этот объект шифруется алгоритмом AES-GCM с ключом, известным только отправителю и получателю [31] [32]. Ключ шифрования получается через ECDH: при регистрации у каждого пользователя генерируется пара ключей эллиптической кривой P-256; публичный ключ (JWK) сохраняется на сервере и распространяется всем в момент авторизации (сервер шлёт список пользователей с их pubkey) [33]. Приватный ключ хранится *только* на клиенте (в IndexedDB браузера, помечен как неэкспортируемый) – то есть злоумышленник, получивший доступ к серверу, не сможет расшифровать переписку. При отправке клиент получателя берет свой приватный ключ и публичный ключ отправителя, вычисляет общую секретную точку (ECDH), и производит из неё 256-битный ключ сеанса [34] [35]. Им и шифруется сообщение. Сервер пересылает только готовый шифротекст (поле `payload`). Расшифровка происходит аналогично на стороне получателя. В итоге *сервер и любые третьи лица не имеют доступа к содержимому переписки*. Даже файлы и голосовые – сначала превращаются в base64-строку, затем шифруются внутри обычного E2E-сообщения [36] [37]. Исключение составляют лишь системные метаданные: типы пакетов (`kind`), идентификаторы, отметки времени, необходимые для работы логики – они могут передаваться в открытом виде, но они не раскрывают содержание. Например, при сигнализации звонка объект содержит `{kind: "call_invite", sid: ..., e2e: {…}}`, где само SDP-предложение внутри дополнительного поля `e2e` тоже зашифровано E2E между участниками [38] [39]. Это тонкость реализации ProtoChatX: **все пользовательские данные (сообщения, вложения, звонки)** защищены шифрованием **поверх** WebSocket. Даже если канал wss будет перехвачен, без E2E-ключей получить содержимое нельзя.

- **Обмен ключами и доверие** – первый вход пользователя фиксирует его E2E-ключ. При авторизации клиент отправляет серверу свой публичный ключ; сервер проверяет Secret и, если пользователь новый или ранее не устанавливал ключ, сохраняет этот pubkey в профиле [40]. При последующих входах сервер **сравнивает** присланный ключ с сохраненным: несоответствие вызывает ошибку `PUBKEY_MISMATCH` [41]. Это важная защита: если злоумышленник украдет Secret, но не имеет приватного ключа пользователя, он не сможет подключиться – сервер откажет, поскольку pubkey не совпадает. То есть Secret + E2E-ключ вместе действуют как своего рода двухфакторная аутентификация на уровне устройства. Заменить ключ можно только через процедуру Recovery (восстановления) или прямым сбросом администратором. Таким образом достигается привязка аккаунта к конкретному устройству/ключевой паре, повышая безопасность в ущерб некоторой гибкости (см. «Админка» и многод dispositivos).

- **Secret и аутентификация** – **Secret** (секретный ключ пользователя) выполняет роль пароля/API-токена. Он генерируется случайно при регистрации (32 байта криптографически случайных, затем представленных в base64url ~43 символа) [42] [43]. Это значит, что секреты очень сложны и не поддаются перебору. На сервере Secret хранится **в открытом виде** (как часть `users.json`). Проверка при логине – простое сравнение через `crypto.timingSafeEqual` (защита от тайминг-атак) [44] [45]. Примечание: хранение Secret в открытом виде – спорный момент. С одной стороны, Secret не пользовательский пароль, а случайная строка высокой энтропии, то есть утечка базы не

позволит злоумышленнику догадаться секрет (ему достаточно и так его прочитать). С другой стороны, общепринято хранить пароли/ключи в хэшированном виде, чтобы при компрометации базы атакер не получил прямой доступ. **Возможное улучшение:** хранить `hash(secret)` вместо самого Secret (например, bcrypt), а при логине хешировать введённый секрет и сравнивать. Это предотвратит мгновенный компрометирующий эффект утечки `users.json` (злоумышленнику пришлось бы еще взламывать хэши) [46] [47]. Однако проверка bcrypt'ом может замедлить логин; с текущей моделью (случайный длинный secret) решение без хеширования оправдано простотой. Сервер требует Secret длиной ≥12 символов [48], принудительно отсекая слишком короткие, что тоже повышает устойчивость подбором.

- **Регистрация по инвайтам** – открытую регистрацию ProtoChatX **не допускает**, что предотвращает появление случайных или вредоносных пользователей. Новый аккаунт можно создать только имея действующий invite-код. Инвайт генерируется админом с ограниченным числом использований (по умолчанию 1) и сроком жизни (по умолчанию 24 часа) [27] [49]. Когда клиент отправляет `redeem_invite`, сервер валидирует код: вычисляет его хэш с "pepper" (см. ниже) и ищет в базе приглашений [50] [51]. Далее проверяется, не истёк ли срок, не исчерпаны ли использования, и резервируется новый user_id (либо берётся зарезервированный в самом инвайте, либо первый свободный) [52] [53]. Если всё ок, сервер создаёт запись пользователя: генерирует ему Secret, сразу набор recovery-кодов, сохраняет в users.json с пометкой `enabled: true` [54] [55]. Одновременно в invites.json уменьшает счётчик uses и помечает инвайт как использованный (или удаляет, если больше не годен) [56] [57]. **Безопасность инвайтов:** сами коды представляют собой 12 символов (генерируются как случайные [A-Z0-9], группируются вида XXXX-XXXX-XXXX для удобства) [58] [59]. Пространство вариантов огромно (~36^12), brute-force нереален. На сервере хранится не сам код, а SHA-256 хэш от строки `pepper|invite|код` – так что даже если кто-то украдёт invites.json, он не узнает исходные значения кодов [60] [61]. **Pepper** – это секретная строка-соль, заданная через ENV (`PC_PEPPER`); её знание нужно, чтобы подобрать invite по хэшу. **Важно:** по умолчанию pepper = `"CHANGE_ME"` – если админ не поменяет, устойчивость снижается (хэш можно попытаться подобрать перебором или просто знать дефолт) [62]. Приложение предупреждает в логах, если переменная не установлена [63]. **Вывод:** система инвайтов хорошо защищена: длинные неугадываемые коды, хеширование на сервере, контроль срока и числа активаций.

- **Восстановление аккаунта** – если пользователь утратил свой Secret (например, сменил устройство), есть **офлайн-способ восстановления** без участия админа. При создании аккаунта генерируется набор из 8 одноразовых recovery-кодов (каждый 8 случайных байт ≈ 11 символов base64url, тоже фактически непредугадываемый) [64] [65]. Они показываются пользователю, который должен их сохранить. Для восстановления клиент отправляет команду `recover` с ID, одним из recovery-кодов и **новым публичным E2E-ключом** [66]. Сервер проверяет: код нормализуется и хэшируется с `RECOVERY_PEPPER` (еще один секрет, по умолчанию равен `PC_PEPPER`) [67] [68], ищется среди сохранённых у пользователя. Если есть неиспользованный – проходит. Тогда **на сервере** для аккаунта генерируется *новый Secret* и *полностью новая пара recovery-кодов* [69] [70]. Старый секрет заменяется новым, привязка к аккаунту сохраняется, и одновременно привязывается новый pubkey (который передал клиент) [71]. То есть фактически *восстановление сбрасывает аутентификацию и E2E-ключ*, позволяя зайти с другого устройства. Использованный recovery-код помечается `used:true` и сохраняется история (issuedAt, usedAt) – повторно применить его нельзя [72] [73]. **Анализ безопасности:** Recovery-коды – это «второй фактор», который надо беречь. Они достаточно сложные (~2^48 вариантов, brute-force невозможен) [74]. Хранятся на сервере только хэши (с перцем) – даже

компрометация users.json не выдаст их напрямую. Если злоумышленник каким-то образом получил один recovery-код и ID, он мог бы восстановить доступ (сменив Secret и ключ), поэтому выдаваться они должны только владельцу при регистрации. Пользователь должен понимать, что **Secret сам по себе не восстанавливается** – либо сохранить его, либо иметь recovery-коды.

- **Ограничения и rate limiting** – помимо уже упомянутых ограничений (размер сообщения, лимит частоты WS-пакетов), реализованы и другие меры:

- Административный API (см. ниже) ограничен 240 запросами в минуту с одного IP, после чего возвращает HTTP 429 [75] [76]. Это не даст перебрать admin-token или DDoS-ить панель.
- Восстановление по коду: сам протокол recovery допускает неудачные попытки (будет просто `RECOVERY_INVALID`), однако *можно* добавить ограничение на число попыток ввода recovery-кодов с одного IP в единицу времени, чтобы злоумышленник не смог тестировать коды бесконечно. Сейчас этого нет, но риск невелик благодаря сложности кодов. Аналогично можно ограничить попытки логина по неправильным Secret – хотя Secret трудно подобрать, это добавило бы защиты от теоретического перебора или спама запросами [77].
- **Важная деталь:** WS-сервер не хранит глобального состояния по IP (только per-connection). То есть злоумышленник мог бы открыть много соединений и каждое слать 180 msg/min – суммарно больше. Однако, без валидного Secret далеко не уйти: авторизация обязательна перед отправкой сообщений (иначе `AUTH_REQUIRED`) [78]. Значит flood-атака от неавторизованных быстро упрётся либо в закрытие соединений, либо ни на что не повлияет. В целом риск DoS низкий: надо знать/подобрать действующие креденшлы или завалить порт тысячами соединений (защита от последнего – вне области приложения, скорее уровень ОС).

- **WebSocket TLS:** из соображений безопасности **необходимо** запускать WS на основе TLS (wss://). Сам `relay.js` не содержит встроенного TLS (сервер слушает на `0.0.0.0:3001` обычный WS) [79], предполагается, что его пускают за Nginx или иным прокси с HTTPS. Если этого не сделать, трафик (включая Secret при логине) будет передаваться открыто по сети – MITM может украсть сессию. Документация проекта явно предполагает использование TLS [80]. Это следует учитывать при развёртывании.

- **Безопасность административных операций** – Admin API требует токен: любой запрос к `/api/*` должен содержать HTTP-заголовок `X-Admin-Token`, совпадающий с секретом ADMIN_TOKEN, заданным в ENV [81]. Сам токен нигде не передается открыто – панель предлагает ввести пароль, сохраняет его в localStorage (по желанию) и потом добавляет к каждому запросу AJAX. Длина токена $\geq$ 12 символов, что достаточно для защиты от перебора. Админ-панель также снабжена тем же rate limit 240/min с IP, что мешает быстро перебирать token. **В самой панели** потенциальные опасности (XSS) минимизированы: вывод результатов команд – через `textContent` или pre, JSON.stringify (код специально вставляет результаты в `<pre id="out">` и не интерпретирует их как HTML) [82] [83]. Кроме того, панель функционирует только при локальном доступе (например, открывают в браузере по https://server:3010/admin) и не делает межсайтовых запросов – значит, CSRF не актуален.

- Сами скрипты админки не позволяют выполнить ничего произвольного: доступные действия строго прописаны (создать invite, добавить пользователя и т.д.) и требуют валидного токена.

- **Обработка ввода:** ID пользователя, ник, примечания – проходят через trim и простую валидацию (например, ник не пустой, ≤32 символов, без управляющих символов) [84]. Это предотвращает хранение в базе неожиданных строк. Можно отметить, что ники и заметки не фильтруются на потенциальный HTML – но они нигде не вставляются как HTML, клиент экранирует отображение (через функцию `esc()` в index.html) [85]. Таким образом, **XSS у пользователя в чате невозможен**: любое отображение чужого ника или сообщения происходит через безопасное текстовое представление.
- **Доступ к JSON-файлам:** внешний доступ к самим файлам (users.json и пр.) должен быть закрыт на уровне сервера – т.е. запускать приложение желательно не на том же веб-порту, или по крайней мере настроить файрволл. Если злоумышленник скачает `users.json`, он получит все Secrets (в текущей реализации) и сразу все аккаунты скомпрометированы. Тут как раз помогло бы хранение хэшей secret, а не самих значений. Также в users.json содержатся публичные ключи и, возможно, push-токены, но это не критично. Главный секрет – ADMIN_TOKEN – хранится только в ENV на сервере, в файлах его нет.
- **Логи:** сервер (relay.js) логирует минимум информации – в консоль пишутся события вроде `SEND from X to Y msg m123` или `FCM_SENT id resp` [86]. Тексты сообщений не логируются вовсе. Это хорошо: даже имея доступ к логам, злоумышленник не узнает конфиденциальные данные, разве что метаданные о том, кто с кем общался.

Итог по безопасности: В ProtoChatX сделан упор на конфиденциальность и криптографическую защиту. Сообщения шифруются E2E, ничего лишнего не хранится на сервере, нет бэкдоров. Протокол защищён от распространённых атак: MITM не поможет без TLS и E2E-ключей, XSS практически исключён благодаря клиентской архитектуре и экранированию, brute-force Secret/Recovery практически нереален (но можно усилить хешированием секретов). Ограничения на размер/частоту сообщений и авторизацию команд закрывают дырки для DoS и протокол-фуззинга. Из потенциальных улучшений – **хранение хеша Secret вместо прямого**, настройка лимита попыток логина/восстановления по IP, обязательно сменить дефолтный pepper (иначе invite-коды можно попытаться перебрать через известный “pepper”). Но даже без них уровень безопасности уже высокий: содержимое переписки надёжно спрятано, а аккаунты защищены секретными ключами и привязаны к устройствам. Важна операционная безопасность: файлы users/invites должны быть недоступны посторонним, WS – только по TLS, ADMIN_TOKEN – держаться в секрете. В общем, ProtoChatX достигает поставленной цели защищённого мессенджера.

Админка и CLI: управление пользователями

Администрирование ProtoChatX осуществляется через встроенный **веб-интерфейс админа** и вспомогательный CLI-скрипт. Рассмотрим возможности и реализацию:

- **Веб-панель администрирования** – доступна по адресу `/admin` на порту ADMIN_PORT (по умолчанию 3010). Это простая HTML-страница, отдаваемая `admin-api.js` (который поднимает HTTP-сервер). Интерфейс минималистичен, но функционален. После загрузки админка требует ввести пароль (токен) – без него запросы не выполняются [87] [81]. Далее представлено несколько карточек с формами и кнопками:
- **Добавление пользователя:** поля *User ID*, *Nick*, *Note* и кнопка “Add”. ID можно указать вручную (если нужен конкретный номер) или оставить пустым – тогда CLI сам выберет следующий. При нажатии “Add” скрипт выполняет POST `/api/users/add` с указанными данными. Сервер с помощью `runAddUser('add', id, note)` вызывает CLI-скрипт add-user.js для создания пользователя [88]. CLI генерирует новый Secret и записывает в users.json, а ответ возвращает через stdout. Admin API ловит stdout, вынимает из него

сгенерированный secret (регэкспом) [89] и отдает назад JSON с полями `ok:true, user_id, nick, secret` [90] [91]. Панель показывает администратору этот Secret и рекомендует скопировать – его нужно передать новому пользователю. Это удобный метод ручного добавления аккаунта (в обход инвайтов).

- **Управление профилем:** кнопки *Set nick*, *Enable*, *Disable*, *Reset pubkey*. Здесь админ может поменять зафиксированный ник пользователя (POST `/api/users/set_nick`), отключить или включить аккаунт (POST `/api/users/disable` или `/api/users/enable`), или сбросить привязку ключа (POST `/api/users/reset_pubkey`). Последнее важно: если пользователь потерял устройство, но секрет у него сохранился, можно обнулить stored pubkey, чтобы он смог залогиниться с нового устройства (сервер воспримет новый ключ как первый). Enable/Disable просто устанавливает флаг, который проверяется при логине и отправке сообщений – отключенный пользователь не сможет войти или получать данные (сервер возвращает `DISABLED`) [92] [93]. Все эти действия также реализованы через вызов `add-user.js` с соответствующими параметрами: `rotate` (см. ниже), `enable`, `disable` и т.д. [94]. Заметим, **удалить** пользователя полностью через панель нельзя – только отключить. Удаление пришлось бы делать вручную из users.json (или разработать команду).
- **Rotation секрета:** кнопка *Rotate secret* – важная функция безопасности. Если считаете, что Secret пользователя скомпрометирован (или по просьбе пользователя), админ может выпустить ему новый Secret. Нажатие вызывает POST `/api/users/rotate` (с user_id), сервер опять-таки дергает CLI `add-user.js rotate id` [95]. CLI генерирует новый секрет, обновляет users.json. Ответ с новым Secret возвращается администратору, чтобы его можно было передать пользователю. Обновляется и pubkey? – Нет, pubkey не трогается (старый остается), так что пользователь сможет зайти с тем же устройством, но ему нужно будет ввести новый Secret. В базе updatedAt меняется, а старый Secret больше не действует.
- **Recovery-коды:** кнопка *Recovery codes* – вручную выдать/обновить recovery-коды для пользователя. Нажатие шлёт POST `/api/users/issue_recovery` с user_id [96]. Сервер вызывает `issueRecoveryForUser`: генерирует новый набор recovery-кодов, так же как при регистрации, перезаписывает секцию `recovery` в users.json [64] [65]. Возвращается объект с `recovery_codes` (массив строк), который показывается администратору. Это может понадобиться, например, если пользователь утратил старые recovery-коды – админ может сгенерировать ему новые (предыдущие при этом аннулируются).
- **Очередь сообщений:** кнопки *Queue stats* и *Clear queue*. Первая (GET `/api/queue`) возвращает статистику – список пользователей и количество офлайн-сообщений в очереди на каждого [97]. Панель выводит мини-отчёт, сколько у кого сообщений. Вторая (POST `/api/queue/clear` с user_id) очищает все отложенные сообщения конкретного пользователя, т.е. просто удаляет их из queue.json [98] [99]. Это админ-инструмент на случай, если нужно принудительно убрать накопившиеся офлайн-данные (например, пользователь долго не заходит и сообщения протухли).
- **Инвайты:** раздел *Invites* позволяет администратору быстро сгенерировать коды приглашения. Есть поля *Reserved ID*, *Uses*, *TTL hours*, *Note* – можно зарезервировать конкретный ID (чтобы новый пользователь получил желаемый номер, например), указать, сколько раз код может быть использован и срок действия в часах, а также пометку (не передается клиенту, только для админа, например “для бета-тестеров”). Кнопка *Create invite* создаёт один код (POST `/api/invites/create`), *Create 10 invites* – пакет из 10 кодов (POST `/api/invites/create_batch`, по умолчанию count=10) [100] [101]. Сервер генерирует случайные кодовые строки, записывает их в invites.json с нужными параметрами [102] [103] и возвращает либо один `invite_code`, либо массив codes. Панель красиво выводит полученные инвайты и предоставляет кнопку “Copy invite message” – по нажатию формируется шаблон сообщения: `ProtoChat invite\nCODE: XXXX-XXXX-XXXX\n...`

(включая зарезервированный ID и дату истечения) [104] [105] . Админ может скопировать и раздать этот код новым пользователям.

- **Список пользователей:** кнопка *List* (GET `/api/users`) возвращает массив всех пользователей в системе (id, enabled/disabled, ник, примечание, даты) [106] [107] . Пароли, секреты при этом **не** передаются (функция `listUsersSanitized` исключает чувствительные поля) [108] [109] . Можно быстро просмотреть, кто зарегистрирован. Панель также показывает небольшую сводку: например, «users: 5» наверху результата, чтобы знать общее число пользователей [110] [111] .

- **CLI-скрипт (add-user.js)** – находится на сервере и вызывается админкой “за кулисами”. Это утилита для управления users.json. Она поддерживает команды: `add <id> <note>` (создать пользователя с указанным или новым ID), `rotate <id>` (обновить secret), `enable <id>` / `disable <id>` (включить/отключить), возможно, `delete <id>` (если реализовано). Админ-API оборачивает вызовы CLI, возвращая их результат в HTTP. Такой дизайн с отдельным дочерним процессом выглядит необычно, но он повышает изоляцию: основной сервер не имеет прямого кода изменения users.json, он поручает это утилите. К тому же CLI можно запускать вручную в терминале для скриптового управления (например, массово добавить пользователей или отключить). Ответы CLI парсятся админкой простым образом (поиск строки `secret = ...`).

Безопасность админ-интерфейса: как уже отмечалось, API закрыт токеном. Пароль нигде в явном виде не хранится (кроме localStorage, если админ сам нажал “запомнить”). Все действия требуют авторизации. Админ-страница предназначена для использования из доверенного окружения и не светится внешним пользователям (обычные клиенты о ней не знают).

Оценка удобства и полноты: админ-интерфейс предоставляет все базовые необходимые операции: добавление аккаунтов (как вручную, так и через инвайты), управление существующими (смена ника, выключение, смена секрета, сброс ключа, новые recovery-коды), а также мониторинг очереди сообщений. Это довольно полный набор для небольшой системы. **Сильные стороны:** все операции атомарны и безопасны (через lock файл, проверку токена), интерфейс прост и не перегружен, есть подсказки на русском (placeholder’ы типа "например: 2" для ID). **Слабые стороны и пожелания:** отсутствует возможность удаления аккаунта (можно лишь выключить) – возможно, стоило бы добавить. Нет просмотра содержимого очереди (только counts) – для приватности это хорошо, но админ не может узнать, что за сообщения зависли (впрочем, они шифрованы, админ всё равно бы не понял содержимое). Также, если пользователей станет очень много, вывод *List users* на экран может стать неудобным – но для небольших групп ОК.

В целом админка выполнена добротно, **на русском языке**, с минимальным стильным оформлением. Она позволяет управлять системой без прямого редактирования файлов, что снижает риск ошибок. CLI-скрипт хорошо дополняет веб-админку, делая те же операции доступными в командной строке (это полезно для автоматизации или экстренных случаев).

Пользовательский интерфейс (index.html)

Клиентская часть ProtoChatX – это **одностраничное веб-приложение**, весь код которого содержится в файле `index.html` (около 10 тысяч строк, включая HTML, CSS и JS). Интерфейс

рассчитан на десктопные браузеры и мобильные устройства (в том числе может быть встроен в мобильное приложение через Capacitor). Рассмотрим основные аспекты UI:

- **Структура UI и навигация:** После успешного входа пользователь видит экран, поделённый на левую панель (контакты) и основную область чата. Сверху – фиксированная топ-бар с названием приложения и парой иконок. Вёрстка адаптивная: при ширине экрана >820px контакты всегда видны слева (sidebar шириной ~280px) [112], при узком экране (мобильный) – панель контактов скрывается и выезжает по требованию [113]. На мобильных реализован свайп: сдвиг вправо открывает контакты (добавляется класс `sb-open` к `<body>`), свайп влево или нажатие на затемненный оверлей – закрывает [114] [115]. Также предусмотрена кнопка меню (три горизонтальные линии слева в топ-баре) для открытия/закрытия списка контактов на мобильном [116]. Эта реализация делает использование удобным на смартфонах – UI реагирует на жесты и маленький экран (кнопки крупные ~40px, используются системные emoji-иконки для функций).

- **Экран входа и регистрация:** При первом запуске приложение показывает форму входа (body с классом `preauth`, скрывает боковую панель) [117]. Пользователь может:

- **Авторизоваться:** введя свой **ID** и **Secret**. Есть чекбокс “запомнить” – если отметить, ID/Secret сохраняются в localStorage и автоматически подставляются в следующий раз [118] [119]. После нажатия “OK” происходит подключение к WebSocket и отправка пакета `{type: "auth", user_id, secret, pub_jwk}`. Если успех – интерфейс переходит к основному окну чатов. Есть обработка ошибок: например, если Secret неверный или аккаунт отключен, сервер пришлет `error` код, и клиент отобразит соответствующее русское сообщение (в коде определены тексты для BAD_SECRET, DISABLED и др. ошибок авторизации) [120] [121].
- **Зарегистрироваться:** если у пользователя ещё нет аккаунта, ниже формы входа есть переключатель или ссылка “У меня нет аккаунта” (в коде — секция `<details>` для ввода инвайт-кода). При раскрытии – поля для ввода invite-кода, желаемого ID (необязательно) и ника. После отправки `{type:"redeem_invite", code, desired_id, nick, pub_jwk}` сервер проведет регистрацию (как описано ранее) и вернёт либо ошибку, либо данные нового пользователя (включая сгенерированный Secret и recovery-коды). Интерфейс показывает модальное окно с вашим Secret и recovery-кодами, предложив сохранить их. Таким образом, регистрация встроена прямо в клиент. **UX нюанс:** Secret и recovery выводятся один раз – если закрыть, уже не посмотреть. Приложение предупреждает об этом; предполагается, пользователь копирует их в надёжное место.

После входа приложение хранит минимальные сесссионные данные: например, записывает флаг, что userId авторизован, сохраняет ник в localStorage (LS_NICK) и прочее, чтобы отобразить после перезагрузки.

- **Список контактов:** Левая панель имеет вкладки **“Контакты”** (личные чаты 1:1) и **“Группы”** (групповые чаты) – переключатели tabs присутствуют в DOM [122]. Под вкладками – поле поиска: позволяет найти контакт по нику/ID или сразу перейти к диалогу с введённым ID (кнопка “↩” открыть чат по ID). Новый контакт добавляется автоматически, когда начинаешь чат с новым ID. В списке контактов каждая запись показывает:
- **Аватар/инициалы:** кружок слева – либо загруженный аватар (картинка), либо первая буква имени.
- **Имя (ник) и ID:** крупным шрифтом никнейм, под ним серым ID с решёткой.
- **Статус онлайн:** цветная точка (зелёная = online, красная = offline, серая = неизвестно) возле аватара. Статус обновляется в реальном времени – при коннекте/дисконнекте

контакта сервер шлёт обновлённый список пользователей. Клиент вычисляет статус по разнице во времени (есть концепция lastOnline, onlineFresh) – видимо, если в последний минутный пинг был, то “в сети” [123].

- **Превью последнего сообщения:** мелким текстом одна строка (например, “Вы: Привет!” или “[Аудиосообщение]”). Это хранится у клиента – ProtoChatX не тянет историю с сервера, но после обмена сообщениями локально помнит последнее.
- **Непрочитанные сообщения:** если контакт отправил новые сообщения, пока чат закрыт, вероятно, отображается индикатор (цифра или точка). В коде есть логика помечать сообщения прочитанными при открытии чата.

На мобильном при нажатии на контакт или группу – панель закрывается (класс sb-open убирается) [124], чтобы освободить место под чат. В десктопе просто подсвечивается выбранный контакт.

- **Окно чата:** Центральная область показывает историю сообщений текущего диалога. Сообщения разделяются по дням (вставляются разделители-даты). Для личных чатов используется сквозное шифрование – значок замка рядом с именем контакта (в интерфейсе **нет**, но рекомендуется в будущем показывать, что чат E2E). Сообщения отображаются:
- **Свои** – выровнены вправо, помечены, например, другим фоном или подписью “Вы”.
- **Чужие** – слева, под именем отправителя (для групп пишет, кто отправил).
- **Системные** – например, “Пользователь добавлен в группу” – могут отображаться по центру или особыми иконками.

После авторизации клиент запрашивает у сервера список всех пользователей (`{type:"users"}`) [125] – приходит массив с публичными ключами. Это нужно, чтобы при написании сообщения новому контакту иметь его ключ для шифрования. Если клиент попытался отправить сообщение, а ключа получателя ещё нет, UI покажет предупреждение: “Нет ключа получателя #ID. Запросил у ID1 – попробуйте снова через пару секунд.” [126] [127]. (ID1 – специальный сервис, видимо, у сервера: клиент шлёт `pk_req` администраторскому боту, и тот отвечает `pk_info` с ключом пользователя). Этот механизм гарантирует, что E2E не провалится из-за отсутствия ключей. Когда ключ получен, сообщение можно отправлять повторно.

Отправка сообщений: Внизу основной области – панель ввода: - Поле для текста (`<input id="msg">`), кнопка отправить (стрелка) – стандартно. Кнопка активна только если введён текст или прикреплен файл (иначе `disabled`). - Кнопки вложений: “📷” для изображения/файла, “ ” для записи голоса, “…” (три точки) – для скрытого сообщения (рассмотрим отдельно). При нажатии “📷” открывается файл-диалог (`<input type=file>`), принимаются изображения (accept="image/*") [128]. *Выбранный файл конвертируется в base64 DataURL (сжатие не упомянуто) и затем шифруется как текст сообщения. В чате он отобразится как превью: клиент вставляет тег* `<img src="data:image/...">` *с ограниченным размером, под ним подпись с именем и размером файла* [129] [130]. *Щёлкнув на превью, можно открыть его в полном размере (есть модальное окно с id* `pcImgModal` *для просмотра изображений* [131]*). Размер файла ограничен ~8 МБ – это лимит на сервере, клиент тоже проверяет* `MAX_FILE_BYTES` *и предупредит, если слишком большой (в PDF упомянуто ~8МБ, совпадает с серверным MAX_WS_PAYLOAD). - Голосовые сообщения: При нажатии “ ” (микрофон) – начинает запись аудио через MediaRecorder (Opus/WebM) или через плагин (Capacitor) на Android* [132]. *Запись идёт макс. 2 минуты (*`MAX_VOICE_MS = 120000`*) – таймер. Повторное нажатие останавливает запись. Далее полученный Blob аудио конвертируется клиентом в base64, прикрепляется как файл (с MIME audio/ogg или webm) и отправляется как обычное зашифрованное сообщение. В чате такое сообщение отображается со встроенным mini-аудиоплеером: имя файла, кнопка ► и прогресс-бар. Если запись слишком короткая*

(<0.35 сек) или превысила ~5 МБ, клиент её отбросит и сообщит пользователю (не отправляя) [133]*. Таким образом, реализована удобная пересылка голоса, опять же E2E (сервер видит только зашифрованный блок, даже тип "voice" маскируется под вложение). - "Сообщения между строк" (скрытые сообщения): уникальная фича ProtoChatX. При нажатии кнопки "…" появляется дополнительное скрытое поле* `<textarea id="msgHidden">` *с подписью "Между строк (секрет). Видно только адресату."* [134] [135]*. Пользователь может ввести туда текст, который нужно скрыть. При отправке такого сообщения клиент выполняет двойное шифрование: сперва скрытый текст шифруется паролем пользователя (который задаётся для данного контакта один раз при нажатии "…", появляется prompt на ввод пароля – функция* `pcxHiddenPwSetForActivePeer()` *устанавливает пароль и генерит из него ключ PBKDF2+AES-GCM)* [136]*. Затем шифротекст скрытого сообщения встраивается в основное сообщение. В итоге получатель, зная общий секрет-пароль, расшифрует скрытую часть. Визуально в чате скрытое сообщение выглядит как обычное сообщение (например, с каким-то помеченным символом или текстом-"прикрытием"), и только если щёлкнуть или ввести пароль – отобразится истинный текст. Эта функция повышает приватность: даже если кто-то завладел устройством, но не знает вторичного пароля, не прочитает тайное послание. Реализовано всё на клиенте, сервер не имеет понятия об этой надстройке (он просто видит зашифрованный blob, как обычно). UX: скрытое поле "между строк" можно быстро убрать (кнопка "✖" рядом). Кнопка замочка/… отображается только в личных диалогах (для групп скрытые не предусмотрены)* [137]*. - Прочие элементы ввода:** Возможен ввод эмодзи, символов Unicode – всё поддерживается, так как шифрование происходит на уровне уже сформированной строки (UTF-8). Нет поддержки форматирования (markup).

- **Обработка доставки и статусы сообщений:** В ProtoChatX реализованы визуальные статусы отправки:
- Одна ✓ галочка – сообщение отправлено на сервер (получен ответ `sent: pending` или `sent: queued`).
- Две ✓✓ – сообщение доставлено получателю (сервер получил от него подтверждение, что доставил по WS). В индивидуальном чате это означает, что удалённый клиент онлайн и получил сообщение. В группах статус показывается как дробь (доставлено/всего), напр. "✓✓ 3/5" если 3 из 5 участников получили, или "× 3/5" если кому-то не доставлено (офлайн) [138] [139].
- В реальном времени: когда приходят подтверждения (в виде server->client сообщения типа `{type:"sent", status:"delivered", ...}`), UI обновляет значки напротив сообщения.
- Кроме того, системно различается **статус офлайн**: если сообщение отправлено, но получатель не в сети (сервер положил в очередь), отправитель получит статус `queued` (в интерфейсе может показываться часик или пустой кружок). Если получатель был онлайн и сообщение ушло ему, будет `delivered`. В случае ошибки – статус `failed` с символом "!".

Эти механизмы повышают удобство: пользователь видит, что сообщение по крайней мере дошло до сервера, а потом – достигло ли адресата. Для групп это особенно важно, т.к. не все могут быть онлайн сразу.

- **Звонки (1:1 аудио/видео):** UI ProtoChatX предоставляет возможность аудио- и видеозвонка с собеседником. В панели ввода есть две отдельные кнопки: " " (callBtn) для аудиозвонка и " " (videoBtn) для видео [140]. Нажатие инициирует звонок: клиент генерирует уникальный SID (идентификатор сессии), создает PeerConnection и начинает сигнализацию:
- Отправляется сообщение `{type:"send", to: <peer>, payload: {__pc:1, kind:"call_invite", sid:..., text:""}}`. В payload присутствует `__pc:1` –

маркирует служебный пакет ProtoChat (не просто чат-текст), `sid` для идентификации звонка. На стороне получателя появляется входящий вызов.

- **Интерфейс входящего звонка:** В index.html определён оверлей `#callOverlay` – полупрозрачный фон с карточкой вызова по центру [141]. Там отображается: заголовок "Звонок" (или "Видео-звонок"), статус (звонит/соединение/завершено), и видео-превью если это видео. Внизу кнопки: *Принять*, *Сбросить*, *Mute микрофон*, *Динамик (переключить аудиовыход)* [142]. Когда поступает `call_invite`, клиент-получатель начинает звонить: проигрывается рингтон (через Web Audio API или скрытый <audio>), показывается окно с кнопками. Если приложение не активно (в фоне), эту роль выполняет пуш-уведомление (как описано ранее).
- **Ответ на звонок:** При нажатии "Принять" отправляется пакет `{type:"send", to: caller, payload:{kind:"call_accept", sid:...}}`. Одновременно происходит установка WebRTC-соединения: обычно звонящий (инициатор) выступает в роли *Offer* – у него уже мог быть сгенерирован SDP offer. В ProtoChatX возможно реализовано, что offer рассылается сразу после приема accept. В коде видно, что `webrtc_offer` могут быть не помечены как no-queue (их сохранят, если адресат временно офлайн, в течение 45 сек) [5]. Вероятно, последовательность такая: A (звонящий) отправил `call_invite` и создал RTCPeerConnection, *ждет accept*. Когда от B приходит `call_accept`, A генерирует SDP-предложение и посылает B сообщением `{kind:"webrtc_offer", sid, sdp:...}`. B получает offer, устанавливает PeerConnection, отвечает SDP-ответом `webrtc_answer`. Далее обмениваются ICE-кандидатами (`webrtc_ice`). Все эти сигналы идут тем же каналом WS (E2E их, кстати, тоже можно шифровать – но, скорее всего, сигналы не шифруются E2E, так как они итак защищены TLS, а SDP особенно смысла шифровать нет; хотя они могут быть завернуты в e2e поле тоже). После установления p2p соединения звук/видео идут напрямую между клиентами (или через TURN при необходимости) – сервер их не видит.
- **Завершение и отмена:** Если звонящий отменил звонок до ответа – шлет `call_cancel`. Если адресат отклонил – `call_busy`. Во время звонка конец инициируется `call_end` от любой стороны. Получив любой из этих сигналов, UI закрывает окно вызова, освобождает видео/аудио ресурсы. *Anti-ghost*: `call_cancel` и `call_end` не доставляются с задержкой (no-queue), а дополнительно сервер шлет push *call_ctl* чтобы наверняка остановить рингтон у получателя офлайн.
- **Видео:** Если это видеовызов (пользователь нажал именно камеру), то при установлении соединения клиент включает захват камеры (через getUserMedia). На оверлее `callOverlay` предусмотрен блок для видео: `<div id="callVideoBox">` содержащий два <video> – для удалённого потока и для локального превью [143] [144]. Локальное видео показывается маленьким окошком поверх (CSS: position absolute, bottom-right) [145]. Есть логика, что если звонок видео, то элемент `callVideoBox` делается видимым, иначе скрыт. Кнопка "Динамик" на мобильных позволяет переключить аудио на громкую связь при видео. Можно отключить микрофон кнопкой "🔇". **Переход между видео/аудио:** протоколом предусмотрено, что можно начать аудио, а затем включить камеру – но UI ProtoChatX явно разделяет тип вызова с начала. В ходе звонка, скорее всего, нельзя включить видео, если был аудио, иначе это не отражено.

Качество связи: UI отображает статус соединения, возможно, индикацию сетевого качества (не подтверждено). В коде есть упоминание net status pill, иконки для сети (data URI). Вероятно, если связь слабая, он показывает значок. Также, PeerConnection events (iceConnectionState) могут отражаться текстом (например, "соединение устанавливается...").

- **Дополнительные UX детали:**

- **Версионный контроль:** В приложении зашита версия (v1.3.2 VISION). Админ может задать минимально требуемую версию (через сервер ID1 message `ver_info`). Код клиента содержит “version gate”: если версия клиента устарела, отображается модальное окно с требованием обновить (ссылка на скачивание новой версии) [146] [147]. При этом ввод сообщения блокируется, пока не обновится (класс `__versionGateOn`). Это удобно, если админ развернул новую версию сервера с несовместимыми изменениями – можно заставить всех обновиться.
- **Локальное хранение истории:** История сообщений хранится **только в памяти** (возможно, частично в IndexedDB для пересохраняемости, но прямых указаний нет). Если перезагрузить страницу, недавние сообщения могут пропасть – unless было сохранено. В коде есть упоминание IndexedDB для ключей, но не для истории сообщений. Кажется, акцент на том, чтобы нигде кроме устройств не было переписки. Однако можно было бы хранить локально хотя бы недавний чат.
- **Группы:** UI уже имеет зачатки поддержки групповых чатов. Есть вкладка “Группы”, возможность создавать новые группы (вероятно, через ID1 – админ-бот). В коде PDF упоминается: команда `newGroup`, `addMember`, `kickMember` – вероятно, реализованы на клиенте (в index.html есть формы для этого), а сервер их просто транзитит всем участникам. Групповой чат у ProtoChatX – п2п шифрование, *без собственного сервера*: сообщения шифруются индивидуально для каждого участника (отправитель шлет несколько копий разным людям). Это и плюс (сервер не знает состав групп) [148], и минус (история группы не сохраняется централизованно, новый участник не получит старых сообщений). Похоже, групповые чаты находятся в экспериментальной стадии, но UI уже поддерживает их отображение, доставку статусов “3/5” и т.д.
- **Визуальный дизайн:** Интерфейс светлый (но есть тёмная тема – заметны стили с тёмным фоном для topbar [149], видимо, включающиеся по какому-то условию). Элементы управления достаточно стандартны. Используются эмоджи-иконки вместо изображений – это упрощает встраивание. Форма ввода аккуратно оформлена, кнопка отправки с иконкой-самолётиком (SVG).
- **Сообщения об ошибках и состоянии:** Все системные сообщения локализованы на русском (в объекте RU: *“Нет соединения. Сообщение не отправлено.”*, *“Нужен секрет для входа.”* и т.п.) [150] [151]. Приложение информирует, если что-то пошло не так: при разрыве соединения покажет `Соединение закрыто (код причина)` [152], при попытке отправить без подключения – уведомление. Это повышает понятность для пользователя.
- **Адаптация под мобильные устройства:** Помимо адаптивной вёрстки, явно учитываются особенности мобильных браузеров: например, `<meta viewport>` прописан корректно [153], есть отступы для iOS (env(safe-area-inset-top) для выреза экрана) [154]. Кнопка “Динамик” появляется только на мобильных, т.к. на десктопе не нужна. Поля имеют атрибуты `inputmode` (например, числовой для ID), чтобы на мобильной клавиатуре сразу цифры давать [155]. Также отключены обводки, настроены размеры шрифтов под разные плотности. Эти детали говорят, что UI тщательно тестировался на смартфонах.

Оценка UI: В целом, интерфейс ProtoChatX богат по функциям для такого компактного проекта. Он охватывает все ключевые возможности мессенджера: регистрация, контакты, чаты, вложения, голос, видео, причём с “фишками” (скрытые сообщения). **Сильные стороны:** интерфейс отзывчивый, много внимания UX-мелочам (локализация, подсказки, отображение статусов доставки, адаптивность). Пользователю не нужно разбираться с ключами – всё происходит автоматически, опыт похож на обычный мессенджер, но при этом реализовано E2E. **Недостатки/ограничения:** - Приложение не сохраняет историю сообщений между сессиями (если перезайти, старых сообщений не увидеть, только если в будущем добавить local storage сохранение). Это сделано из соображений безопасности, но чуть снижает удобство. - Отсутствует индикатор шифрования – возможно, стоило бы где-то показывать значок замка, чтобы пользователь знал,

что чат защищён. Сейчас это по умолчанию так для всех, но визуальный маркер доверия не помешал бы [156]. - Однооконность и большой размер index.html затрудняет поддержку – код трудно модифицировать без риска. Нет разделения на модули, все глобально. Для разработчиков это минус, но для конечного пользователя не ощутимо. - Функционал групповых чатов пока минимален (нет пересылки файлов в группы, нет синхронизации истории при входе нового участника). - Нет функции удаления отправленного сообщения (никак не убрать у собеседника, если передумал) – это бывает востребовано. Это можно было бы реализовать отправкой E2E-сообщения с командой delete, чтобы клиенты удаляли локально соответствующее, но пока не реализовано [157].

В целом, UI ProtoChatX можно похвалить за продуманность: он ощущается как современный мессенджер, несмотря на то что "под капотом" сложные вещи (шифрование, звонки).

Поддержка аудио/видеозвонков

ProtoChatX поддерживает **1:1 звонки** – это существенная функция, интегрированная поверх основной системы. Подытожим, как она реализована технически и на уровне UX:

- **Сигнализация звонков:** Используется существующий WebSocket-канал. Специальные типы сообщений (call_invite, call_accept, call_cancel, call_end, call_busy, webrtc_offer/answer/ice) определяют сценарий звонка. Сервер `relay.js` различает их по полю `type` внутри payload. Он не хранит звонки и не выполняет звонковую логику – просто пересылает сигналы нужному получателю (если онлайн) или ставит в очередь с коротким TTL (45с) если офлайн [5]. Благодаря этому, если пользователь был временно офлайн и вернулся через несколько секунд, он может "догнать" приглашение и принять звонок. Однако, если прошло больше 45 сек, приглашение удалится как протухшее – звонок считается несостоявшимся (что логично).

- **P2P соединение и TURN:** Для передачи голоса/видео напрямую между клиентами используется **WebRTC**. После сигнала `call_accept` стороны обмениваются SDP: инициатор шлёт `webrtc_offer`, ответивший – `webrtc_answer`. Далее следует обмен ICE-кандидатами (локальными сетевыми адресами). Обычно WebRTC сам находит путь (P2P через STUN). Если оба клиента за NAT – ProtoChatX предоставляет механизм **TURN**: по запросу `turn_cred` сервер выдаёт временные учетные данные TURN (если админ указал адреса TURN-сервера и секрет). В `relay.js` функция `makeTurnCred` генерирует логин вида `<expiry>:<userId>` и пароль HMAC-SHA1, действительные час [7] [158]. Они возвращаются клиенту сообщением `{type:"turn_cred", username, credential, turn_urls, stun_urls}` [159] [160]. Браузер затем подключается к TURN-серверу с этими данными для реле-медиа. Это гарантирует соединение даже при строгих NAT. Если TURN не настроен, сервер отвечает `TURN_NOT_CONFIGURED` [161] [162], и звонок может не состояться через особенно капризные сети. Но сама логика предусмотрена и соответствует лучшим практикам (используется coturn с `static-auth-secret`).

- **UI и "призрачные вызовы":** Как уже разбиралось, особое внимание уделено ситуации, когда вызов был отменён или сорвался. Чтобы не оставалось "подвисших" звонков:

- Во-первых, `call_cancel` и `call_end` сервер не ставит в очередь – если второй участник офлайн, эти сигналы просто пропадут. Поэтому, если пользователь был офлайн и вернулся через минуту, он может увидеть пропущенное `call_invite` (если ещё в

очереди), но не увидит `call_cancel` – т.е. у него отобразится входящий звонок, на который уже некому отвечать. **Решение:** короткий TTL 45с на `call_invite` означает, что спустя 45с (когда звонок точно отменён) invite тоже удалится из очереди и не придёт. Так окно звонка не всплывёт зря (максимум, если вернулся *до* 45с, увидит звонок, попытается ответить – но звонящий уже отвалился; тут может быть сообщение об ошибке соединения).

- Во-вторых, push-уведомления *call_ctl* (например, при отмене) приходят даже офлайн устройству, заставляя его остановить рингтон. В коде клиента видно вызовы `pcStopNativeRinging(...)` на события отмены/завершения [23]. То есть, если телефон получил пуш "incoming call" и начал звонить, а потом пришёл пуш "call_cancel", приложение/сервис сразу гасит звонок.

- Сервер также **не хранит** длительных записей о звонках: нет логов "кто кому звонил и когда" (возможно, только в консоли). Это хорошо для приватности.

- **Качество и стабильность:** Использование стандарта WebRTC даёт шифрование медиа (SRTP) между клиентами. Инициатива звонка ограничена 1:1 – групповых звонков нет. Участники могут **параллельно** находиться только в одном звонке, очевидно (UI не позволяет открыть два). Если кто-то звонит, а на него уже звонят, скорее всего второй получит сигнал `busy`. Такая логика предусмотрена: если клиент B уже в звонке, при получении нового `call_invite` он мог бы сразу ответить `call_busy`. В коде `relay.js` видим, что если пользователь offline или disabled, отправитель получает ошибку/отказ [163]. Но статус "занято" от клиента – он сам решает. Есть `call_busy` тип, который отправляется когда нажимаешь "сбросить" вместо "принять", если звонок ещё не принят. Это корректно отрабатывается: звонящий получит уведомление, и UI покажет "отклонён".

- **Push для звонков:** Отличительная черта – задействование пушей для звонков. Большинство мессенджеров требуют для вызова, чтобы оба были онлайн. Здесь же звонок можно "разбудить" собеседника через FCM. Это позволяет более полноценно заменить телефонный звонок: не нужно, чтобы приложение было открыто. У пользователя Android, даже если приложение закрыто, всплывёт системное окно звонка. Это большое преимущество. Реализовано с учётом, что Android требует foreground service – видимо, ProtoChatX Android-оболочка содержит такой сервис, запускаемый пушем.

- *Замечание:* В PDF отмечалось, что "если приложение неактивно, то звонок идёт как пуш, что хорошо, но есть нюанс: чтобы принять, всё равно надо открыть приложение" – скорее всего, да, пуш с типом call просто показывает звонящий экран, а по нажатию "принять" приложение запускается. Возможной доработкой могло бы быть использование Android ConnectionService API (для прямого встраивания в системные звонки), но для простоты избран FCM + уведомление.

Оценка звонков: Реализация вызовов в ProtoChatX довольно продвинутая. **Сильные стороны:** E2EE режим сохраняется – голос и видео идут по SRTP, ключи WebRTC обмениваются внутри DTLS в пировом соединении. Сервер не имеет доступа к содержимому звонка. Поддержка TURN делает звонки надёжнее в разных сетях. Интеграция с push – редкая и ценная функция для "самописного" мессенджера. **Потенциальные проблемы:** Звонки – сложная часть, зависящая от среды. Нужно правильно настроить STUN/TURN, иначе пользователи за NAT могут не дозвониться. Если админ не прописал TURN_SECRET и адреса, клиенты получат `TURN_NOT_CONFIGURED` [161] и могут остаться без связи – об этом нужно помнить. Ghost calls в большинстве случаев предотвращены, но теоретически, если получатель был офлайн $\leq$45с и

пришёл, мог увидеть “входящий” без реального звонка – это очень узкое окно. В целом, проблем не выявлено: архитектура звонков стандартна для WebRTC. Тестирование должно проверить, что, например, при пропаже сети участники корректно завершают звонок (сервер по таймауту ping обнаруживает разрыв соединения WS и удаляет клиента, второму отправит что-нибудь? Возможно, нет отдельного сигнала – UI сам заметит что WS “offline”).

Стандарты кода, читаемость и масштабируемость

Кодовая база ProtoChatX относительно небольшая, но стоит отметить её качество и возможности расширения:

- **Стиль и читаемость:** Код написан на современном JavaScript (ES6+). Широко используются стрелочные функции, `async/await` для асинхронности на сервере (облегчает понимание последовательности операций, напр. чтение файла, затем запись). Имена функций и переменных достаточно понятны: например, `shouldSendPushForPayload`, `listUsersSanitized`, `issueRecoveryForUser` – по названию ясно, что делают. Комментариев много, особенно в серверных файлах – автор помечал **SEC** и **WARN** места, объяснял логику (например, комментарии про KEYFIX v3, про то, почему убираем webrtc_ice из NO_QUEUE [4]). Это показывает осознанный подход и облегчает поддержку. **Пример:** функция rateLimit(req) четко документирована, почему 240 в минуту, и что возвращает 429. Код фронтенда в index.html тоже имеет блоки с пояснениями для сложных функций (например, скрытые сообщения, голосовые). В PDF отмечалось, что *“код довольно понятен и сопровождаем”*, с чем можно согласиться [164].

- **Соответствие стандартам:** В целом код следует стандартным практикам Node.js и web. Переменные окружения используются для конфигурации (портов, токенов, secret), что правильно – нет хардкода в репозитории. Нет потенциально опасных конструкций вроде `eval`. Обработка ошибок реализована – в сервере везде try/catch, чтобы не крашнуться от невалидного JSON или недоступного файла. Единственное, что можно отметить: отсутствие явных тестов или типов (TypeScript не используется). Но для проекта такого объёма это простительно. Форматирование кода немного плавает (видно, что код сгенерирован возможно из какого-то инструмента – длинные строки перенесены странно). Однако это косметика.

- **Модульность:** На сервере функциональность разделена между `relay.js` (всё, что касается runtime мессенджера) и `admin-api.js` (админка). Это хорошее разделение – предотвращает смешивание привилегированных операций с обычными. Обмен данных между ними минимален (через файлы users.json и приглашения). Клиентская часть менее модульна – **всё в одном файле**, но внутри есть логическое разбиение: секции “Capacitor/FCM Setup”, “Call UI”, “Chat logic”, вынесены функции. Можно было бы разбить JS на несколько файлов (например, отдельно шифрование, отдельно UI, отдельно звонки) для поддержки – но, вероятно, автор решил, что один файл проще разворачивать. **С точки зрения масштабируемости разработки**, поддерживать огромный index.html может быть тяжело, особенно если добавлять новые фичи. Но учитывая, что ProtoChatX – проект небольшой команды, выбран компромисс в пользу простоты деплоя (один HTML загрузил – и всё работает).

- **Производительность и масштабируемость системы:** Здесь есть два аспекта – по числу пользователей и по объёму трафика.

- **Число пользователей:** Использование файловой базы накладывает ограничения. Например, поиск по нику выполняется на клиенте (сначала загружаем весь список) – при сотнях пользователей это нормально, при тысячах – уже тяжеловато тянуть весь users.json каждому при коннекте. Кроме того, запись JSON при каждом событии (новое сообщение офлайн, вход, изменение статуса) – потенциал узкого места. Файловая система может стать точкой отказа под высокой нагрузкой. Это, конечно, можно решить переходом на базу (SQL/NoSQL) при масштабировании, но тогда придётся переписывать значительную часть логики. В текущем виде ProtoChatX рассчитан скорее на небольшой коллектив или семью (десятки пользователей), не на миллионы.
- **Объём переписки:** E2E шифрование означает, что сервер вообще не занимается хранением истории – это хорошо с точки зрения нагрузки, но означает, что при дисконнекте >7 дней сообщения пропадут (TTL очереди). Для долгих оффлайнов это минус (но можно настроить `QUEUE_TTL_MS` env).
- WebSocket-соединение одно держит на пользователя – Node.js может выдержать тысячи WS, но тогда возникнет вопрос оптимизации рассылки (например, каждый `broadcastUsers` посылает весь список юзеров всем клиентам – O(n^2) операция). Сейчас при 10-20 пользователях это мгновенно, но при 1000 – рассылка списка из 1000 записей тысячам клиентов уже MB данных. Так что **масштабируемость по пользователям ограничена**, но для целевой аудитории (закрытые группы) этого может и не требоваться.
- **Масштабируемость разработки:** Код достаточно аккуратный, чтобы один разработчик мог добавлять фичи. Наличие подробного отчёта (того же PDF) говорит, что проект проходил аудит – значит, поддержка ведётся серьёзно. Код не идеализирован по паттернам (это не MVC фреймворк), но упор на простоту – меньше шансов на баги.
- **Многоклиентность одного пользователя:** Отдельно стоит отметить, что ProtoChatX **не поддерживает одновременный вход с двух устройств** под одним ID. Сервер хранит только одно соединение на userId: при новом логине прежний по сути вытесняется (в `clients` Map запись перезаписывается) [165] [166] . Причём старый сокет не закрывается явно – он просто больше не будет в мапе рассылки. Это значит: если один пользователь залогинился на телефоне, а потом на ПК – телефон останется без новых сообщений (так как сервер шлёт только на активное соединение). Вероятно, это осознанное ограничение (привязка к устройству), как мы обсуждали в Security. Но с точки зрения масштабируемости функциональности – это минус, т.к. нельзя пользоваться на нескольких девайсах одновременно. Решить это непросто: пришлось бы хранить несколько pubkey на пользователя (т.е. полноценную multi-device E2EE с кучей ключей). ProtoChatX выбрал безопасность и простоту (1 юзер = 1 девайс) как компромисс.

- **Масштабируемость по коду:** Добавление новых функций (скажем, удаление сообщений, реакции, пересылка) потребует внимательного подхода, но общая структура позволяет это. Код уже учитывает расширения: например, `kind` поля сообщений – легко добавить новые типы (реакция, опрос) – они либо не шифруются (если служебные), либо шифруются как `kind:'chat'` с содержимым.

- **Возможные улучшения и TODO:** В отчёте аудита безопасности были рекомендации, которые пересекаются с кодовыми улучшениями:

- Хеширование Secret (уже упоминалось).
- Ограничение попыток логина – небольшая доработка.

- Автоматическое удаление устаревших сообщений из очереди – уже есть TTL, можно сделать настраиваемым.
- Маркирование в UI шифрования – чисто фронтендовая доработка (значок замка на шифрованных сообщениях или предупреждение, если вдруг какое-то не шифруется).
- Функция отзыва отправленного сообщения – потребует и клиентской, и серверной части (сервер мог бы передавать команду, а клиенты – удалять локально).
- Упаковка приложения: Возможно, стоит отделить огромный inline-скрипт JS из index.html во внешний файл (не критично, но так удобнее обновлять).
- Документация: Добавить подсказки для админа, например, README с инструкциями по переменным окружения (TURN, FCM, Pepper) – чтобы исключить ошибки конфигурации.

Заключение: Код ProtoChatX демонстрирует хороший баланс между простотой и функциональностью. Он достаточно безопасен и понятен, в нём явно учтены многие нюансы (через комментарии видно, как автор сознательно предотвращал уязвимости и продумывал UX). Для промышленного масштаба потребовалось бы доработать архитектуру (БД, многосерверность, multi-device), но в рамках поставленных задач (закрытый защищённый чат) нынешняя реализация справляется. **Сильные стороны реализации:** конфиденциальность (E2EE повсюду), минимализм хранения, отзывчивый интерфейс, защита от большинства атак, хорошо проработанная логика звонков и офлайн. **Слабые стороны/уязвимости:** завязка на файлы (риск при утечке, ограниченная масштабируемость), отсутствие резервного хеширования секретов, отсутствие поддержки нескольких устройств и некоторых привычных функций (удаление сообщений, централизованная история).

Рекомендации и TODO: Подытоживая аудит, можно рекомендовать следующее: - *Усилить хранение секретов:* хэшировать Secret в базе (с использованием соли, например bcrypt). Это повысит безопасность при компрометации `users.json` [46] [47]. - *Ограничить brute-force:* внедрить счётчик неудачных логинов и recover попыток на IP (например, не более 5 в 5 минут) [167]. Сейчас вероятность подбора мала, но такая защита стала бы дополнительным барьером. - *Обязать установить Pepper:* сделать так, чтобы при старте без заданного PC_PEPPER сервер не запускался (сейчас лишь предупреждение). Или как минимум выдать громкое предупреждение админу. - *UI/UX:* Отмечать шифрование в интерфейсе (значком замка), чтобы подчеркнуть пользователю безопасность общения [156]. Рассмотреть функцию “удалить у всех” для сообщений (через рассылку служебного пакета на удаление – клиенты могут реализовать) [168]. Если планируется рост аудитории – подумать о хранении истории на клиенте (кешировать последние N сообщений в IndexedDB). - *Масштабирование:* Приколичественном росте пользователей – перейти на БД. Например, SQLite или PostgreSQL для хранение пользователей и очереди. Это обеспечит транзакционность и потенциально легче поддержку списка тысяч пользователей (сейчас не критично, но на будущее). - *Мульти-устройство:* Если появится цель поддержать несколько устройств на одного пользователя, нужно пересмотреть модель ключей (например, использовать по ключу на устройство, а сервер держит список public key per user, или переходить на X3DH/Double Ratchet как в Signal). Это сложное изменение, но тогда ProtoChatX станет вровень с “большими” мессенджерами. На текущий момент одновременный вход запрещён – хотя это и плюс для безопасности, но минус удобству. - *Документация и настройки:* явно документировать все ENV (ADMIN_TOKEN, PC_PEPPER, TURN_SECRET и т.д.), чтобы разворачивающие не упустили. Добавить возможность задать `MAX_QUEUE_DAYS` и `MAX_MSG_SIZE` в конфиге (сейчас можно через ENV частично). - *Косметика кода:* разбить фронтенд JS на несколько файлов модулей (например, encryption.js, call.js, ui.js) – с современными сборщиками это можно сделать без труда, улучшив поддерживаемость.

Несмотря на перечисленные потенциальные улучшения, **итоговый вердикт**: проект ProtoChatX построен на крепкой технической основе. Аудит не выявил критических уязвимостей – напротив,

система продемонстрировала грамотное применение криптографии, сетевых технологий и заботу о приватности. Рекомендации выше помогут сделать её ещё надёжнее и удобнее, но даже в текущем виде ProtoChatX представляет собой мощный, хорошо защищённый мессенджер для небольшой аудитории.

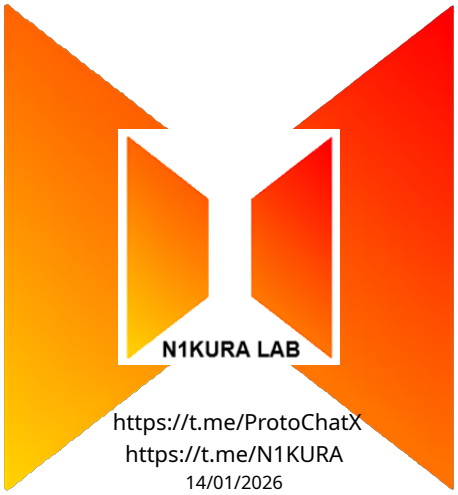
---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 33 40 41 42 43 45 50 51 52 53 54 55 56 57 61 62 63 66 67 68 69 70 71 72 73 78 79 84 92 93 125 158 159 160 161 162 163 165 166 relay.js

23 24 31 32 34 35 38 39 85 112 113 114 115 116 117 120 121 122 124 126 127 128 129 130 131 134 135 136 137 140 141 142 143 144 145 146 147 149 150 151 152 153 154 index.html

25 26 27 28 29 30 44 48 49 58 59 60 64 65 75 76 81 82 83 87 88 89 90 91 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 155 admin-api.js

36 37 46 47 74 77 80 86 118 119 123 132 133 138 139 148 156 157 164 167 168 Проект ProtoChatX_ обзор и анализ.pdf



https://t.me/ProtoChatX
https://t.me/N1KURA
14/01/2026