Протокол VK RPC

Таблица версий документа

30 января 2024	Первая версия
11 февраля 2024	Уточнения в формате пакетов
06 ноября 2024	Добавлен формат RPC поверх формата пакетов
26 января 2025	Уточнил детали формата сериализации RPC ошибок
30 января 2025	Зафиксировали минимальную длину криптоключа
18 мая 2025	Уточнение поведения диагональных запросов. Уточнение поведения таймаутов.
29 мая 2025	Уточнение поведения таймаутов.
31 июня 2025	Модификации для поддержки TL2
19 июня 2025	Добавлена информация про UDP транспорт
23 ноября 2025	Фиксирование версии документы

Feedback приветствуется, vk:@hrissan @udp2

Общие сведения

VK использует собственный протокол **RPC** с собственными транспортами, реализованными поверх **TCP** и **UDP**.

Транспорт обеспечивает установление и разрыв соединение, шифрование и надёжную доставку в обе стороны сообщений, которые состоят из тела некоторой ограниченной длины плюс отдельно 32-битного типа сообщения.

Для шифрования и аутентификации используется общий секрет (крипто ключ).

А протокол **RPC** реализован в терминах обмена сообщениями.

Транспорты поддерживают сообщения длиной до 4 гигабайтов, но на уровне **RPC** размер искусственно ограничивается примерно 16 мегабайтами.

*Streaming не реализован, сообщения в **UDP** доставляются в произвольном порядке.*

RPC

Начнём с высокоуровневого протокола, а транспорты опишем ниже

Соединение

Обмен сообщениями происходит в соединении. Соединение устанавливается при первой необходимости отправить сообщение (либо заранее) и обычно существует, пока не разорвётся одной из сторон. В случае **TCP** это реальное соединение, в случае **UDP** соединение существует в виде структур данных (контекста соединения) в транспортах клиента и сервера.

В случае **TCP** клиент используется рандомный исходящий порт, и есть механизм **keep-alive** который позволяет быстро (20 секунд с настройками по-умолчанию) закрывать старые соединения в случае отсутствия активности.

В случае **UDP** клиент использует фиксированный исходящий порт, механизм **keep-alive** внешний (пинговалки), а существующее соединение закрывается, как только успешно завершается **handshake** нового соединения с того же порта и **IP** адреса. (Детали этого описаны в разделе **UDP Transport**).

Запросы и ответы

Клиент, то есть сторона, которая установила соединение, является инициатором **RPC** запросов. На каждый полученный запрос сервер отправляет один ответ, сразу или через некоторое (возможно долгое) время. У каждого запроса есть опциональный таймаут, выбираемый клиентом. Если таймаут не установлен, он будет бесконечным.

Запросы идентифицируются с помощью **query_id** (64-битное знаковое значение), который уникально выбирается клиентом в рамках соединения. Нулевой **query_id** запрещён для удобства реализации (использования значения как флаг), сервер получив такой запрос, должен ответить ошибкой. Рекомендуется начать со случайного положительного **query_id**, увеличивать его на 1 при каждого следующем запросе, и при достижения максимального значения перейти обратно к 1.

Благодаря **query_id** сервер может присылать ответы в произвольном порядке, по мере их готовности. Также клиент по **query_id** может отменить запрос, который ему больше не нужен. Это важно для **long poll** запросов (о них ниже).

Сервер не отправляет клиенту запросов, требующих ответа, а только уведомления, которые ответа не требуют. Эта асимметрия не случайна и не должна нарушаться в следующих версиях протокола, так как делает реализацию fixed memory гарантий на сервере чрезмерно сложной.

Типы сообщения, используемые RPC.

rpcInvokeReq	0x2374df3d	Запрос, от клиента к серверу
rpcReqResult	0x63aeda4e	Ответ (успешный или с ошибкой), от сервера к клиенту
rpcReqResultError	0x7ae432f5	Ответ с ошибкой, устаревший! Не должен отправляться сервером, но пока что должен парситься клиентом. В случае ошибки сервер должен отправить rpcReqResult с ошибкой внутри.
rpcCancelReq	0x193f1b22	Отмена запроса, уведомление от клиента к серверу
rpcServerWantsFin	0xa8ddbc46	Начало протокола graceful shutdown, уведомление от сервера к клиенту, что нужно перестать отправлять запросы в этом соединении
rpcClientWantsFin	0x0b73429e	Уведомление от клиента к серверу в рамках протокола graceful shutdown, что больше запросов не будет

Уведомления должны доставляться транспортом (в UDP это требует нового флага в пакете) только после всех сообщений, отправленных до этого, так как полагаются на порядок доставки.

TODO - реализовать в протоколе пользовательские уведомления rpcServerNotify

Запрос (сообщение rpcInvokeReq)

Отправляется клиентом, содержит **query_id** для идентификации запроса и ответа.

```
[0x2374df3d] [query_id] [message]
```

Системные заголовки, оборачивание

Если необходимо передать дополнительные параметры, используется оборачивание клиентского запроса в один или более системный заголовков.

```
[0x2374df3d] [query_id] [header] [message]
```

```
[0x2374df3d] [query_id] [header] [header2] ... [message]
```

Все заголовки являются сериализованными структурами **TL** в **boxed** формате, где первые 4 байта - уникальный **magic**. Предполагается, что и клиентское сообщение имеет формат **TL** и сериализовано в **boxed** формате.

Код **RPC** после чтения **query_id** смотрит в цикле на следующие 4 байта, и если они совпадают с известным ему **magic** системного заголовка, парсит системный заголовок. Цикл останавливается, когда встретит неизвестные ему 4 байта. Остаток запроса передаётся в пользовательский обработчик, который уже разбирает его так, как считает нужным.

Client **magic** не должен совпадать ни с одним **magic** системных заголовков. В **VK** это обеспечивается тем, что все клиентские запросы и все системные заголовки собираются в файле **combined.tl**, где при совпадении **magic** генератор **TL** выдаст ошибку.

Если сообщение клиента имеет формат, отличный от **TL**, например **JSON**, то для исключения коллизий рекомендуется добавлять 4 нулевых байта в качестве префикса сообщения перед отправкой, и убирать после получения, нулевой **magic** запрещён в **TL** и поэтому не совпадёт ни с одним системным заголовком, даже определённым в будущем.

Системные звголовки запроса.

rpcDestFlags	0xe352035e	Дополнительные параметры запроса (**Extra**, см. Определение в файле **combined.tl**)
rpcDestActor	0x7568aabd	**Actor_id**, которому предназначен запрос. Используется rpc proxy для направления в нужный кластер
rpcDestActorFlags	0xf0a5acf7	Комбинация rpcDestActor и rpcDestFlags
rpcProxy.diagonal	0xbceb94c3	Указание прокси отправить запрос в одну реплику каждого шарда кластера
rpcProxy.diagonalTargets	0xee090e42	Указание прокси отправить запрос в каждую реплику заданного диапазона шардов кластера
rpcTL2Marker	0x29324c54	Указание, что тело запрос в формате TL2. Этот заголовок должен быть последний, сразу перед телом запроса.

Запрещается указывать более одного **Actor_id** или более одного **Flags** в одном запросе (то есть, например, два **rpcDestActor**, два **rpcDestActorFlags** или **rpcDestActor** и **rpcDestActorFlags**). Если сервер обнаруживает такую ошибку, он обязан не пытаться обработать запрос, а вернуть ошибку пользователю.

Устаревший ответ с ошибкой (сообщение rpcReqResultError)

Содержит query_id для идентификации ответа, код и описание ошибки. Не должен больше никогда отправляться серверами, так как не позволяет передать дополнительные параметры, вместо этого нужно использовать обычный ответ (см. ниже).

```
[0x7ae432f5] [query_id] [code] [description]
```

Ответ (сообщение rpcReqResult)

Отправляется сервером, содержит query_id для идентификации ответа.

```
[0x63aeda4e] [query_id] [message]
```

Системные заголовки, оборачивание

Если необходимо передать дополнительные параметры, используется тот же принцип, что и при запросе - оборачивание ответа в один или более системный заголовков.

```
[0x63aeda4e] [query_id] [header] [message]

[0x63aeda4e] [query_id] [header] [header2] ... [message]
```

Также в случае ошибки, вместо сообщения от обработчика ответ будет закончен системным заголовком с кодом и текстом ошибки

```
[0x63aeda4e] [query_id] [header] ... [0xb527877d] [code]
[description]
```

Утверждается, что из-за бага в vkext вместо такого формата ошибки движками отправляется другой, содержащий ещё раз query_id (который должен игнорироваться).

```
[0x63aeda4e] [query_id] [header] ... [0x7ae432f5] [query_id] [code]
[description]
```

Всего получается 4 варианта ответа, все реализации должны уметь парсить все варианты, пока не придём к каноническому виду, которым выбран reqError, как самый простой и не содержащий двусмысленностей.

По факту код в C++ репозитории на 18 мая 2025 не умеет парсить формат reqError. Все остальные парсеры парсят все 4 формата.

Мы решили, что после реализации парсинга reqError в коде C++ и раскатки RPC proxy в

production, все реализации перейдут на возврат этого варианта, как канонического.

Системные звголовки ответа.

reqResultHeader	0x8cc84ce1	Дополнительные параметры ответа (**Extra**, см. Определение в файле **combined.tl**). Перед отправкой маске полей в этом объекте делается AND с аналогичной маской Extra запроса, так что не будут сериализованы и отправлены поля, которые клиент не запросил явно (например, потому что не знал об их существовании). Это обычный для TL механизм расширения.	Должен отправляться, если flags != 0. Может (неоптимально) отправляться и если flags == 0
reqError	0xb527877d	Содержит код и описание ошибки, не отправляется серверами из-за ошибки в vkext	Должен отправляться после переходного периода
rpcReqErrorWrapped	0x7ae432f6	Содержит код и описание ошибки. Вообще непонятно зачем нужен и кем используется.	Не должен отправляться никогда
rpcReqError	0x7ae432f5	Содержит код и описание ошибки, плюс избыточный query_id, отправляется серверами из-за ошибки в vkext	Должен отправляться во время переходного периода

Как и в случае запросов, запрещено указывать более 1 заголовка каждого типа в одном ответе.

Запросы и ответы в формате TL2

Для минимизации усилий при переходе на TL2 мы не меняем структуру запросов и ответов. Только тело запроса оборачивается в специальный маркер **rpcTL2Marker**. При этом сервер не готовый к TL2 воспринимает это как неизвестный запрос и отвечает ошибкой. Сервер, готовый к TL2 вызывает соответствующий обработчик.

Формат тела ответа должен соответствовать запросу, если тело запроса в TL2, то и тело ответа в TL2, то же самое и с TL1.

При проксировании запросов и ответов формат тел не меняется. Предполагается, что инфраструктурный код всех прокси и серверов будет достаточно быстро обновлен для поддержки нового формата заголовок. Формат TL2 сохраняет возможность для прокси прочитать “первое поле” запроса, поэтому схемы шардирования first int, first string и

аналогичные продолжают работать эффективно. Поддержка схем шардирования в формате TL2 будет выкатываться постепенно, по мере необходимости. С целью перейти на формат TL2 везде, где это возможно небольшими силами.

Таймауты запросов

При запросе клиент может передать таймаут в поле **timeout_ms** объекта **extra**. Нулевое значение таймаута считается эквивалентом неустановленного. Если клиент выбирает таймаут, он должен также установить локально таймер, и отменить запрос локально при его срабатывании с ошибкой “клиентский таймаут” (-3000). Сервер, получив запрос с неустановленным либо слишком большим таймаутом, выбирает таймаут по своему усмотрению. Сервер должен установить таймер не каждый запрос, и при его срабатывание ответить клиенту ошибкой “серверный таймаут” (-4000) (кроме long poll запросов, о которых ниже).

TODO - уточнить коды ошибок клиентского и серверного таймаута. Сделать один и тот же код ошибки таймаута, чтобы гонки не влияли на семантику.

В качестве оптимизации сервер может применять неточные таймеры, (**timer bins**, etc), так как таймер на сервере нужен только чтобы сэкономить ресурсы сервера на уже ненужных запросах, и не влияет на семантику.

Для запросов к движкам традиционно используется таймаут в 270 миллисекунд.

Для **long poll** запросов рекомендуется таймаут от 1 до 5 минут. При таком таймауте типичные 100к клиентов при отсутствии каких-либо событий на сервере создают паразитную нагрузку от 1500 до 300 RPS.

Если таймаут longpoll запроса на сервере почти прошёл, но условия срабатывания longpoll не реализовалось, рекомендуется ответить клиенту неким NOP-ответом, например пустым списком событий, etc. Таким образом при нормальной работе механизма не будет появляться ошибок таймаутов, а ошибки будут индикатором каких-то проблем. Также если клиент подписан на некий поток событий по фильтру, сервер может в NOP-ответе передать новый offset общего потока событий, чтобы сэкономить существенные ресурсы на повторной фильтрации со старого offset.

Рекомендуется отправлять NOP-ответ за 5-10 секунд до установленного клиентом таймаута, а если клиент установил слишком маленький таймаут, то вообще не отправлять NOP-ответ, в надежде, что клиент, увидев ошибки, увеличит таймаут.

32-битное беззнаковое поле **timeout_ms** позволяет выразить таймаут до 50 суток, что более, чем достаточно в обозримой перспективе. Гранулярность таймаута в 1 миллисекунду тоже достаточно для всех сценариев использования протокола.

Мы используем именно таймаут, а не deadline, так как установка часов с точностью до миллисекунды и поддержание их в таком состоянии является невыполнимой для нашей инфраструктуры задачей. Поэтому время, которое запрос проводит от отправки rpc клиентом до получения rpc сервером не учитывается сервером при установке таймера, так что серверные таймеры имеют тенденцию срабатывать чуть позже клиентских. Попытки компенсировать этот эффект некоторыми эмпирическими формулами представляются бессмысленными.

Отмена запроса (сообщение rpcCancelReq)

Если клиент отменяет запрос, который уже был отправлен в соединение по любой причине, кроме срабатывания локального таймера (в этом случае на сервере сработает такой же таймер и поэтому отправка необязательна), клиент может отправить серверу в это соединение сообщение **rpcCancelReq**.

```
[0x193f1b22] [query_id]
```

Необходимость отправки связана с тем, что существуют долгие или очень долгие запросы (механизм **long poll**, когда сервер задерживает ответ на клиентский запрос до наступления некоторого условия, например, клиент передаёт **offset** последнего известного ему события в некотором журнале, а сервер отвечает новыми событиями, только когда они появятся). Выполнять запросы, отменённые клиентом, уже не нужно, поэтому их ранняя отмена позволяет экономить ресурсы сервера.

При отмене запроса локально клиентом ответ уже может быть в пути, поэтому клиенты должны игнорировать ответы с неизвестными **query_id**, а серверы должны игнорировать отмены запросов с неизвестным **query_id**.

При разрыве соединения, все контексты запросов в этом соединении автоматически отменяются, как на клиенте, так и на сервере.

Реализация отмены на прокси.

Обычно прокси входящие запросы добавляет в словарь по входящему **query_id** и входящему **connection_id**. А также в словарь по паре исходящего query_id и исходящего **connection_id**.

Если клиент отменяет свой запрос к проксе, то прокси должна отменить соответствующий ему свой исходящий запрос. Если клиент разрывает соединение, то прокси должна отменить все исходящие запросы полученные через это соединение. Такая несложная логика гарантирует, что любой **long poll** запрос (либо запрос, отменённый клиентом по некоторому событию) правильно и быстро отменяется через любое количество прокси при любых событиях отмены либо разрыва соединений.

Протокол завершения (graceful shutdown)

Если сервер желает перезапуститься, то при разрыве соединения в нём могут оказаться как летящие запросы, так и ответы, в итоге клиент получит одну или более ошибок. Чтобы избежать этого, существует специальный протокол закрытия сервера.

Сначала сервер перестаёт устанавливать новые соединения, затем сообщает всем клиентам о том, что желает закрыться, отправляя в каждое клиентское соединение один раз уведомление **rpcServerWantsFin**.

```
[0xa8ddbc46]
```

Клиент, получив это сообщение, обязан не начинать отправку новых запросов, завершить отправку того запроса, который уже начал отправляться, и после этого отправить уведомление **rpcClientWantsFin**.

```
[0x0b73429e]
```

После этого клиент ждёт, пока сервер пришлёт все ответы, и затем закрывает соединение.

Сервер же, получив **rpcClientWantsFin**, перестаёт принимать новые запросы (при получении запроса считает нарушением инварианта и немедленно закрывает соединение), отменяет все long poll запросы с ошибкой (-3000, серверный таймаут), по возможности отменяет обычные выполняющиеся запросы, если это поддерживает **API** (в golang делается Cancel() контексту, переданному обработчику), и дожидается закрытия соединения клиентом после получения тем всех ответов.

Клиент также может сам инициировать протокол завершения соединения, отправив в соединение **rpcClientWantsFin**. Дальнейшие действия по закрытию выполняются в точности так же, как описано выше для случая, когда инициатором являлся сервер.

Если у клиента нет ни одного отправленного в соединении запроса, то клиент должен сразу закрыть соединение, без отправки **rpcClientWantsFin**. Это важная оптимизация для случая, когда тысячи клиентов подключены к тысячам серверов.

Мотивация

Протокол с явным **rpcClientWantsFin** выбран потому, что клиент не знает, что какие-то его запросы оказались long poll запросами (это зависит от состояние клиента и сервера). А потому не может просто так ждать, пока получит на все свои запросы ответы, ответов может не быть сколь угодно долго. Поэтому нам нужно явное уведомление от клиента, что запросов в этом соединении больше не будет. Таким уведомлением мог бы быть **TCP FIN**, но во-первых это бы не работало для **UDP**, во-вторых, после **TCP FIN** в соединении не

работает механизм пинг-понг, а значит время завершения было бы ограничено временем до отправки пинг, что может быть меньше желаемого.

Специфика для TCP

Инициатором закрытия **TCP** соединения выбран клиент, так что при нормальном срабатывании протокола сокеты в состоянии **TIMED_WAIT** на сервере не накапливаются.

Также в случае **TCP** сервер начинает с закрытия Listen-сокета, так что новая копия сервера может немедленно запуститься, открыть этот сокет и начать принимать новые соединения.

Специфика для UDP

В случае **UDP**, пока сервер не завершился полностью, новая копия сервера не может быть поднята и принимать соединения, так как нет простого способа отфильтровать пакеты для старых и новых соединений. Поэтому в случае **UDP** сервер полагается на то, что перед началом протокола завершения трафик будет снят с него помощью пинговалки. Поэтому сервер не отправляет rpcServerWantsFin, а просто отказывается подключать новых клиентов на время завершения. А клиенты при получении сообщения от пинговалки о том, что сервер недоступен, сами инициируют протокол завершения.

TCP Transport

Общие сведения

TCP обычно используется для общения между компонентами на одной машине, например **kphp** и **rpc-proxy**, но может использоваться и между разными машинами (компоненты статсхауса).

В качестве нижележащего транспорта для протокола используется **TCP** или **Unix stream socket**.

Шифрование опционально, и будет включено если любая из сторон требует шифрования.

Есть встроенный механизм **Keep-Alive**.

Forward Secrecy будет реализована начиная с версии протокола 2, так что все реализации есть смысл довести до версии 2, затем запретить любые версии меньше 2.

Все числа сохраняются в виде **Little Endian**.

Соображения о дизайне протокола будут использовать курсив. Мы решили сначала минимально изменить существующий протокол для решения самых насущных проблем, а дизайн нового, более сложного протокола отложить до лучших времён.

Формат сообщения

Сообщение состоит из заголовка, тела, контрольной суммы и выравнивания.

Заголовок.

```
type packetHeader struct {
        length uint32
        seqNum uint32
        tip    uint32
}
```

В качестве длины в заголовке используется длина содержимого + 16. Обычно пользователи транспорта устанавливают лимит на длину сообщения, которую они согласны читать, популярный лимит, которым пользуется RPC это 2^24-1 байт (лимит длины тела на 16 байтов меньше).

Лимит нужен как защита от атаки, когда отправка заголовков с большой длиной заставит сервер выделить много памяти под тела сообщений.

Номер первого сообщения равен `uint32(-2)` и затем увеличивается с каждым следующим сообщением.

Тип сообщения.

Nonce	0x7acb87aa	Установка соединения, не рекомендуется для использования клиентами
Handshake	0x7682eef5	Установка соединения, не рекомендуется для использования клиентами
Ping	0x5730a2df	Механизм **Keep-Alive**, не может использоваться клиентами
Pong	0x8430eaa7	Механизм **Keep-Alive**, не может использоваться клиентами
Пользовательский	*	Свободно используется клиентами

Фактически здесь неудачно зарезервировано 2 значения из 4 пользовательских байтов под нужды транспорта. Лучше было бы зарезервировать специальный бит в заголовке.

Содержимое

После заголовка идут байты содержимого. Они интерпретируются только для типов сообщений, обрабатываемых на уровне транспорта.

Контрольная сумма

Контрольная сумма считается от всех байтов заголовка и тела. В качестве контрольной суммы сначала используется **CRC32**, но стороны могут договориться об использовании более эффективной **CRC32C** с помощью флага в сообщении **Handshake**.

Контрольная сумма коротковата и плоховата, но решили пока не менять её. Она играет роль MAC - единственное, что защищает от интерпретации рандомных байтов отправленных в зашифрованное соединение атакующим.

Выравнивание

Если шифрование выключено, выравнивание не делается.

Выравнивание делается только после включения шифрование, первое сообщение, после которого делается выравнивание - сообщение **Handshake**.

Сначала добавляются нулевые байты до размера сообщения, кратного 4.

В реализации рекомендуется проверять эти байты на 0.

Затем, если отправителю нужно завершить криптоблок **AES**, он может добавить от 0 до 3 значений `uint32(4)`.

Так как минимальный размер сообщения (с пустым телом) равен 16 байтов, значения выравнивания `uint32(4)` при чтении заголовка сообщения можно отличить от длины и пропустить.

Установка соединения

Сначала клиент и сервер обмениваются сообщениями **Nonce**, которые содержат достаточно информации для установки версии протокола и способа шифрования.

После успешного обмена сообщениями **Nonce** обе стороны выводят общий ключ шифрования.

Затем клиент и сервер обмениваются сообщениями **Handshake**

После этого клиент и сервер могут посылать друг другу любые сообщения согласно их договорённостям.

Сообщение Nonce

Номер этого сообщения -2, тип 0x7acb87aa, поле “длина” заголовка сообщения **Nonce** должно быть меньше 1024 байта, в противном случае участник должен закрыть соединение.

Содержимое сообщения

```
type nonceMsg struct {
        KeyID       [4]byte
        Encryption byte
        Version     byte
        Flags       uint16
        Time        uint32
        Nonce       [16]byte
        DHPoint     [32]byte
        . . .
}
```

Сообщение **Nonce** может содержать в конце дополнительные поля из следующей версии протокола, неизвестные участнику. Такие поля должны игнорироваться.

KeyID - идентификатор ключа

Это просто первые 4 байта крипто ключа, если ключ короче 4 байтов, то дополняется нулями до 4 байтов. Раскрытие этих байтов не представляет проблемы, так как длина ключа может быть довольно большой. Ключи с идентификатором равным 4 нулям запрещены. Минимальная длина ключа 32 байта. Клиент и сервер должны отказываться соединение с неподходящими ключами.

Encryption - необходимость шифрования

Со стороны клиента

0	Клиент согласен работать только без шифрования
1	Клиент согласен работать только с шифрованием
2	Клиент согласен на оба варианта, сервер выбирает вариант
*	Остальные значения интерпретируются сервером так же, как 2

Со стороны сервера

0	Сервер выбрал работать без шифрования
1	Сервер выбрал работать с шифрованием
*	Остальные значения интерпретируются клиентом как ошибка, клиент обязан закрыть соединение

Клиент и сервер ориентируются на свойства самого соединения при выборе этих параметров. При использовании **Unix Socket** а также **TCP Socket**, у которого оба адреса являются адресами локальной машины возможна работа без шифрования.

Также если оба адреса **TCP Socket** находятся в одной доверенной подсети, то тоже возможна работа без шифрования. Это имеет смысл при установке соединений контейнер-контейнеровоз или контейнер-контейнер. А также внутри одного дата центра.

Если участник требует шифрования, а партнёр предлагает работать только без шифрования, то участник закрывает соединение.

Version - версия протокла

Клиент ставит максимальную версию, которую он поддерживает. Сервер отвечает минимумом от версии клиента и своей максимальной поддерживаемой версии. Любая сторона может отказаться работать со старыми версиями исходя из требований безопасности и закрыть соединение.

Flags - флаги

Игнорируются и должны устанавливаться в 0 обеими сторонами.

Time - время (Unix timestamp)

Клиент и сервер обязаны закрыть соединение, если их время отличается больше, чем на 30 секунд в любую сторону.

Nonce

Клиент и сервер обязаны генерировать эти байты с помощью криптографического генератора случайных чисел.

Маловато случайных байтов, каждый участник может полагаться на случайность только своих 16.

DHPoint (только в версии протокола 2)

Открытый ключ **Diffie-Hellman** согласно протоколу **X25519** (https://rfc-editor.org/rfc/rfc7748.html)

Если клиент предлагает только схему без шифрования, то может быть заполнен нулями для экономии ресурсов.

Если сервер выбрал схему без шифрования, то может быть заполнен нулями для экономии ресурсов.

Отправляя сообщение Nonce версии 2 можно заставить сервер делать нетривиальную работу по вычислению открытого ключа Diffie-Hellman (скорость этого примерно 10000 в секунду на ядро) до того, как клиент подтвердит, что знает крипто ключ.

Чтобы избежать этот, можно добавить раунд в протокол, клиент присылает доказательство (например sha2-256(server.Time server.Nonce crypto_key)) того, что знает крипто ключ в сообщении Nonce2, сервер только потом генерирует и посылает открытый ключ D-H. Это замедлит установку соединения на 1 ping-pong. DHPoint нельзя добавить в Handshake, так как это сообщение тогда не будет иметь forward secrecy, к тому же придётся менять шифрование потока на лету, что сложно реализовать.

Включение шифрования

Если стороны договорились об использовании шифрования, то обе стороны выводят общие ключи шифрования из обоих сообщений **Nonce**, своего и партнёра.

Используется **AES-256** в режиме **CBC**. Для каждого из двух направлений общения выводится свой ключ и вектор инициализации.

Шифрование блоками, первый блок шифрования начинается с первого байта сообщения **Handshake**. Если записан неполный блок, и участнику протокола нужно немедленно отправить сообщение, он дополняет сообщение до полного блока значениями `uint32(4)`, как описано в разделе выравнивание.

Версия протокола 0

Сначала формируется crypto init message M, путём конкатенации

Server Nonce	16 bytes
Client Nonce	16 bytes
Client Time	4 bytes
Server IP	4 bytes (Внимание, Little Endian, как и все другие поля)
Client Port	2 bytes
Направление	6 байтов. Зависит от направления отправки, если от клиента к серверу, то “CLIENT”, а от сервера к клиенту “SERVER”
Client IP	4 bytes (Внимание, Little Endian, как и все другие поля)
Server Port	2 bytes
Crypto Key	Зависит от длины крипто ключа
Server Nonce	16 bytes, используется второй раз
Client Nonce	16 bytes, используется второй раз

AES ключ выводится путём конкатенации первых 12 байтов `md5(M[1:])` и `sha1(M)`
AES вектор инициализации выводится, как `md5(M[2:])`

Тест векторы:

Server Nonce	ABCDEFGHIJKLMNOP
Client Nonce	abcdefghijklmnop
Client Time	0x01020304
Server IP	0x0d0e0f10
Client Port	0x090a
Направление	SERVER
Client IP	0x05060708
Server Port	0x1112
Crypto Key	hren

Для направления отправки от клиента к серверу

Crypto init message	414243444546474849 4a4b4c4d4e4f506162636465666768696a6b6c6d6e6f7004030201100f0e0d0a09434c49454e540807060512116872656e414243444546474849 4a4b4c4d4e4f506162636465666768696a6b6c6d6e6f70
AES ключ	28b5a5313b3ea9e2f6f0293e0748b2f743b0e112779faa77a3ee9d71ae70dda6
Вектор инициализации	80387128489168b336d998762bce6fef

Для направления отправки от сервера к клиенту

Crypto init message	414243444546474849 4a4b4c4d4e4f506162636465666768696a6b6c6d6e6f7004030201100f0e0d0a095345525645520807060512116872656e414243444546474849 4a4b4c4d4e4f506162636465666768696a6b6c6d6e6f70
AES ключ	e3cf8557ea4ad963c3b637d466388403841d2e989a1fc684ac691c44b05ac9bb
Вектор инициализации	1efd4c8aa43a87d1ea5488a1bc669269

Версия протокола 1

Как в версии протокола 0, но при формировании crypto init message вместо Client IP, Client Port, Server Port используются нулевые значения, а вместо Server IP ставится Server Time, которое забыли в прошлой версии протокола.

Тест векторы:
Исходные значения такие же, как в предыдущей секции, Server Time совпадает с Server IP и равен 0x0d0e0f10.

Для направления отправки от клиента к серверу

Crypto init message	414243444546474849 4a4b4c4d4e4f506162636465666768696a6b6c6d6e6f7004030201100f0e0d0000434c49454e540000000000006872656e414243444546474849 4a4b4c4d4e4f506162636465666768696a6b6c6d6e6f70
AES ключ	373374076f52d8f6bb5b063f17b9eb9fb4194e429cf02e207300add4c28a8e57
Вектор инициализации	cea8f827019de36741f73e5948aea5be

Для направления отправки от сервера к клиенту

Crypto init message	414243444546474849 4a4b4c4d4e4f5061626364656667686 96a6b6c6d6e6f7004030201100f0e0d0000534552564552000000000000 6872656e41424344454647484 94a4b4c4d4e4f5061626364656667 68696a6b6c6d6e6f70
AES ключ	3ce0c95487d99754688e0508a036c8c02727f297d0311db6273d69 c07ac7a0d2
Вектор инициализации	34411262ac3e172bc1a2d086b4f1ecb5

Версия протокола 2

Как в версии протокола 1, но к crypto init message конкатенируется выведенный общий секрет **Diffie-Hellman**, 32 байта.

Тест векторы:

Протокол Diffie-Hellman

Скаляр клиента	012344abcdefghijklmnopqrstuvwxyz
Скаляр сервера	567899ABCDEFGHIJKLMNOPQRSTUVWXYZ
Точка клиента	4b7fe2cd2aa7067de1d46b7aeced9ca5fc748748c324855d1f83a97 72da45d49
Точка сервера	c0d54fe02bae5a4336105769a99a128db969ac8c034334ec201f6b 6016635a56
Общий секрет	4541d9fd5263298736d6ecdfa8c5834e12b54e2ad3bb95a50d2085 dd4075f458

Код генерации на C

```c
#include <stdio.h>
#include <openssl/evp.h>

void print_hex(unsigned char * ptr, size_t len) {
   for (size_t i = 0; i < len; i++) {
      printf("%02x", ptr[i]);
   }
   printf("\n");
}
```

```
EVP_PKEY * point_from_scalar(const char * role, unsigned char * point, unsigned
char * scalar) {
   printf("%s PRIVATE KEY: ", role);
   print_hex(scalar, 32);

   EVP_PKEY *pkey = EVP_PKEY_new_raw_private_key(EVP_PKEY_X25519, NULL, scalar,
32);

   size_t point_len = 32;
   if (EVP_PKEY_get_raw_public_key(pkey, point, &point_len) <= 0 || point_len
!= 32) {
      printf("client pubkey get failed\n");
   }

   printf("%s PUBLIC KEY: ", role);
   print_hex(point, 32);
   return pkey;
}

void shared_from_peer(const char * role, EVP_PKEY * pkey, unsigned char *
shared_secret, unsigned char * peer_point) {
   EVP_PKEY *peerkey = EVP_PKEY_new_raw_public_key(EVP_PKEY_X25519, NULL,
peer_point, 32);

   EVP_PKEY_CTX *ctx = EVP_PKEY_CTX_new(pkey, NULL);
   if (!ctx) {
      printf("CTX is empty");
   }
   if (EVP_PKEY_derive_init(ctx) <= 0) {
      printf("EVP derive initialization failed\n");
   }

   if (EVP_PKEY_derive_set_peer(ctx, peerkey) <= 0) {
      printf("EVP derive set peer failed\n");
   }

   size_t shared_len = 32;
   if (EVP_PKEY_derive(ctx, shared_secret, &shared_len) <= 0 || shared_len !=
32) {
      printf("Shared key derivation failed");
   }
   printf("%s shared secret: ", role);
   print_hex(shared_secret, 32);
}

int main() {
   EVP_PKEY *clientkey = NULL;
   EVP_PKEY *serverkey = NULL;
```

```
    unsigned char client_scalar[] = "012344abcdefghijklmnopqrstuvwxyz";
    unsigned char server_scalar[] = "567899ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    unsigned char client_point[32] = {};
    unsigned char server_point[32] = {};

    clientkey = point_from_scalar("client", client_point, client_scalar);
    serverkey = point_from_scalar("server", server_point, server_scalar);

    unsigned char client_shared[32] = {};
    unsigned char server_shared[32] = {};
    shared_from_peer("client", clientkey, client_shared, server_point);
    shared_from_peer("server", serverkey, server_shared, client_point);

    return 0;
}
```

Для сборки нужно написать

```
$> gcc x25519.c -lcrypto -o x25519
```

Для направления отправки от клиента к серверу

Crypto init message	4142434445464748494a4b4c4d4e4f5061626364656667686 96a6b6c6d6e6f7004030201100f0e0d0000434c49454e54000000000000006872656e4142434445464748494a4b4c4d4e4f5061626364656667686 96a6b6c6d6e6f704541d9fd5263298736d6ecdfa8c5834e12b54e2ad3bb95a50d2085dd4075f458
AES ключ	c513a88366728c719ffe885d943b0faa701ff7f0b061311b9af5fa5a0ec830ef
Вектор инициализации	bbaf9484282c1d021c21d9da05e822c0

Для направления отправки от сервера к клиенту

Crypto init message	4142434445464748494a4b4c4d4e4f5061626364656667686 96a6b6c6d6e6f7004030201100f0e0d0000534552564552000000000000006872656e4142434445464748494a4b4c4d4e4f5061626364656667686 96a6b6c6d6e6f704541d9fd5263298736d6ecdfa8c5834e12b54e2ad3bb95a50d2085dd4075f458
AES ключ	987d9938b0ea97bae1604e78d47131a5b0dc426054d5f9423d14f867480dce1d
Вектор инициализации	cf55ffd9615629f9cc7fc6b14d9a48f8

Сообщение Handshake

После того, как стороны включили шифрование либо договорились работать без него, отправляется сообщение **Handshake**. В основном он нужен для того, чтобы убедиться, что ключи **AES** вывелись одинаково, а также чтобы обменяться секретными настройками протокола, которые нельзя отправлять открытым текстом в сообщении **Nonce**.

Номер этого сообщения равен -1, тип 0x7682eef5, поле “длина” заголовка сообщения **Handshake**, так же как и у сообщения Nonce должна быть меньше 1024 байта, в противном случае участник должен закрыть соединение.

Если номер, тип, или длина сообщения оказались невалидны, то с высокой вероятностью `2^-(32+32+22)` это означает, что клиент и сервер вывели ключи шифрования по-разному, чаще всего из-за того, что ключи имеют одинаковый **KeyID**, но дальше различаются. Сервер должен напечатать понятное сообщение об ошибке в этом случае, иначе будет очень сложно догадаться, что идёт не так.

Содержимое сообщения

```
type handshakeMsg struct {
        Flags     uint32
        SenderPID NetPID
        PeerPID   NetPID
        . . .
}
type NetPid struct {
        Ip      uint32
        Port    uint16
        Pid     uint16
        Utime   uint32
}
```

Сообщение **Handshake** может содержать в конце дополнительные поля из следующей версии протокола, неизвестные участнику. Такие поля должны игнорироваться.

Flags - флаги

```
flagCRC32C = 0x00000800 //RPC_CRYPTO_USE_CRC32C
```

Если Клиент поддерживает **CRC32C**, он ставит этот флаг в своём сообщении. Если клиент поставил флаг, и сервер тоже поддерживает **CRC32C**, он тоже ставит этот флаг в своём сообщении.

Если сервер поставил этот флаг, то и клиент и сервер включают **CRC32C** начиная ссообщения, следующего за **Handshake** сообщением.

SenderPID, PeerPID

Некоторые реализации заполняют поля действительными значениями Uptime, Pid, IP и Port. Эти значения никак не используются самим транспортом, но могут использоваться клиентами, как ключи различных метрик.

На Pid выделено всего 16 бит, хотя было бы лучше 32.

Keep-Alive

Каждый участник протокола должен иметь таймаут на чтение полностью следующего сообщения. Если за таймаут считан хотя бы один байт, но не полное сообщение, соединение закрывается. А если за таймаут не считано ни одного байта, то участник отправляет сообщение **Ping** с возрастающим номером, сбрасывает таймаут и снова начинает читать следующее сообщение, ожидая довольно быстро получить в ответ **Pong**. Если таймаут срабатывает повторно, то участник закрывает соединение.

Таймаут может быть произвольный и зависеть от свойств соединения, например отсутствовать или быть больше, если подключение по **Unix Socket**. Если таймаут фиксированный в реализации **PacketConn**, то не клиенте рекомендуется таймаут немного меньше, чем на сервере, чтобы сервер и клиент не становились одновременно инициаторами **Ping**.

Рекомендуемые таймауты на клиенте 10 секунд, на сервере 11.

Ping и **Pong** запрещены, пока стороны не обменялись сообщениями **Nonce**, **Handshake**. Поэтому таймаут на установку соединения равен двум таймаутам на чтение сообщения.

Любой участник может установить меньший таймаут на установку соединения, и разрывать связь, если за этот таймаут не произошёл полный обмен сообщениями **Nonce**, **Handshake**. Такая настройка может понадобиться на серверах, которые часто атакуют.

Типы сообщений 0x5730a2df для **Ping** и 0x8430eaa7 для **Poing**, содержимое сообщений одинаково.

```
type pingPong struct {
        ID uint64
```

```
}
```

Размеры содержимого этих сообщений должны быть равны строго 8 байтов.

Участник, получив сообщение **Ping** с каким-то ID, должен ответить сообщением **Pong** с этим же ID.

Участник не должен отправлять следующий **Ping**, пока не получит ответ на предыдущий.

Участник, получив сообщение **Pong**, которого не ожидает, должен закрыть соединение.

UDP Transport

Общие сведения

UDP транспорт используется для общения с движками на других машинах и межсервисного общения.

UDP протокол позволяет иметь один сетевой сокет на машине для общения со всеми, в отличие от **TCP**, где для каждого соединения нужен был свой сокет, и была проблема, что могло быть большое количество соединений, каждое из которых использовало мало памяти сокетного буфера.

Проблема ненадёжности **UDP** решается нумерованием сегментов сообщений и отправкой **ack**-ов — подтверждений получения сегмента.

Шифрование используется всегда — даже при локальном трафике.

Механизм **keep-alive** реализован отдельным инфраструктурным компонентом - ping-engine (пинговалкой), в котором хранятся все **UDP** эндпоинты в инфраструктуре.

В целом протокол содержит огромное количество странностей в дизайне, делающий корректную реализацию максимально проблематичной, особенно что касается установки соединений и ротации ключей. Криптостойкость протокола всерьёз не изучалась, но она не отвечает современным требованиям как минимум из-за коротких Connection ID и использования CRC32 вместо криптографического хэша. Также протокол крайне неэффективен по размерам заголовков, лишних 60-120 байтов в каждой датаграмме.

Мы не видим смысла развивать существующий протокол, а собираемся сделать новый транспорт поверх DTLS. Существующий же протокол документируется в процессе

clean-room реализации на актуальных для нас языках/рантаймах. В том числе проводится изучение и исправление ошибок в старой реализацией, которой пользуются движки, по мере того, как улучшается наше понимание протокола и тесты.

Формат UDP сообщения

RPC сообщения, отправляемые поверх **UDP** транспорта бывают надёжные и ненадёжные. Надёжные сообщения доставляются с **exactly-once** гарантией — вторая сторона eventually получит каждое ровно один раз. Ненадёжные сообщения не имеют гарантий — каждое будет получено ноль или один или сколько угодно раз.

Каждое надёжное **RPC** сообщение разбивается на одну или несколько частей. Они нумеруются (каждая часть имеет **seq-num**, уникальный для всего соединения), начиная с 0. Это позволяет отправлять **ack**-и — подтверждения получения частей.

Каждая часть одного **RPC** сообщения доставляется в отдельной **UDP** датаграмме, две части одного сообщения никогда не бывают в одной датаграмме. Части разных **RPC** сообщений могут быть в одной датаграмме, но только имеющие подряд идущие **seq-num**-ы.

Одна **UDP** датаграмма состоит из следующих частей:

- **Unencrypted** заголовок
- **Encrypted** заголовок
- полезная нагрузка из одной или нескольких частей разных **RPC** сообщений
- **padding** из 0, 4, 8 или 12 нулевых байт
- повторный **Unencrypted** заголовок
- **crc32** контрольная сумма

Из всего этого только первый **Unencrypted** заголовок и **crc32** отправляются не шифрованные, все остальные части шифруются. Их суммарный размер должен быть кратен 16 (особенности шифра) и для этого существует **padding**. Его достаточно, так как длины **RPC** сообщений уже кратны 4-м байт, и их части тоже нарезаются кратными 4-м, а также длины всех заголовков кратны 4-м.

Контрольная сумма берётся от всего остального содержимого датаграммы.

Unencrypted Заголовок

`netUdpPacket.unencHeader#00a8e945`

```
  flags:#
  local_pid:flags.0?%net.Pid
  remote_pid:flags.0?%net.Pid
  generation:flags.0?#
  pid_hash:flags.2?long
  crypto_flags:flags.3?#
  crypto_sha:flags.4?%True
  crypto_random:flags.5?%(Tuple # 8)
  encrypted_data:flags.7?%True
 = netUdpPacket.UnencHeader;
```

Unencrypted заголовок служит сейчас для двух целей — определить, в какое соединение пришла датаграмма (либо создать новое) и определить параметры для дешифрования.

Идентификация соединения (сессии) происходит либо по тройке local_pid, remote_pid и generation, либо по pid_hash. Одновременно присылать и то, и то сейчас запрещено.

В crypto_flags и crypto_random лежит информация о шифровании пакета. crypto_random — это вектор инициализации для дешифровки датаграммы **AES-256** шифром.

В crypto_flags проставляются в реализации на **go** всегда, а в **C++** только в датаграммах с pid в **Unencrypted** заголовке. Нулевой бит отвечает за то, что шифрование производится через **AES-256**, первый бит за то, что это handshake сообщение. Также, с 8-го бита по 20-й невключительно лежат первые 12 бит общего crypto ключа (key_id). Транспорт, получив key_id, обязан найти среди своих ключей подходящий, если же такого нет, то не обрабатывать датаграмму.

Флаги crypto_sha и encrypted_data должны быть выставлены всегда.
TODO — выяснить, правда ли это.

Encrypted Заголовок

```
netUdpPacket.encHeader#251a7bfd
  flags:#
  time:flags.9?int
  version:flags.10?int
  packet_ack_prefix:flags.13?#
```

```
packet_ack_from:flags.14?#
packet_ack_to:flags.14?#
packet_ack_set:flags.15?%(Vector #)
packet_num:flags.20?#
packets_from:flags.21?# packets_count:flags.21?#
prev_parts:flags.22?#
next_parts:flags.23?#
prev_length:flags.24?#
next_length:flags.25?#
single_rpc_msg:flags.26?%True
multiple_rpc_msgs:flags.27?%True
zero_padding_4_bytes:flags.28?%True
zero_padding_8_bytes:flags.29?%True
packet_offset:flags.30?long
window_control:flags.31?#
= netUdpPacket.EncHeader;
```

Encrypted заголовок содержит всю информацию о присланных частях **RPC** сообщений, **ack**-и на полученные той стороной части сообщений, время, версию и **padding**.

packet_ack_* - подтверждения

Поля с подтверждениями получения некоторых частей **RPC** сообщений:

- packet_ack_prefix подтверждает получения всех пакетов с номерами [0..packet_ack_prefix]
- packet_ack_from и packet_ack_to подтверждает получения всех пакетов с номерами [packet_ack_from..packet_ack_to]
- packet_ack_set просто содержит список полученных частей.

packet_num, packets_from, packets_count - присланные части

Датаграмма может содержать либо только packet_num, либо только пару packets_from и packets_to, либо ни одного из этих полей — например, если сторона решила только прислать **ack**-и.

packet_num показывает, что весь payload датаграммы является цельным **RPC** сообщением. Если он равен -1, значит прилетело unreliable сообщение. На них **ack**-и не отправляются.

packets_from и packets_count указывают, что в payload датаграммы сериализовано несколько частей **RPC** сообщений, от packets_from и до (packets_from + packets_count) невключительно.

Первая часть может содержать не целое **RPC** сообщение, а только его часть, равно как и последняя часть может содержать только часть **RPC** сообщения. Строго промежуточные части обязательно содержат целые сообщения, так как в одной датаграмме запрещено передавать две части одного сообщения.

Чтобы определить, являются ли первая и последняя части самодостаточными или нет, используются флаги prev_parts и next_parts.

prev_parts, next_parts - склейка частей

prev_parts сообщает, сколько предыдущих (по **seq-num**) частей из потока нужно склеить с первой присланной в payload-е частью, чтобы получить цельное **RPC** сообщение. Например, если прилетела датаграмма с packets_from=10 и prev_parts=4, значит цельное сообщение будет состоять из конкатенации четырёх частей - 7, 8, 9 и 10.

Аналогично, next_parts показывает, сколько следующих частей из всего потока нужно объединить с последней частью в payload-е, чтобы получить целое сообщение. Например, датаграмма с packets_from=10, packets_count=6 и next_parts=3 говорит, что части 15, 16 и 17 образуют единое **RPC** сообщение.

Также, если в датаграмме всего одна часть и оба флага > 0, значит это промежуточная часть **RPC** сообщения, не начало и не конец. Например, при получении датаграммы с packets_num=25, prev_parts=2 и next_parts=1 понимаем, что нужно склеить части 23, 24, 25 и 26.

prev_length, next_length, packet_offset - fixed memory гарантия

prev_length сообщает длину префикса **RPC** сообщения до первой части в датаграмме. Семантика как у prev_parts, только в байтах, а не в частях. Аналогично, next_length сообщает длину суффикса сообщения после последней части.

Эти поля были добавлены во 2-й версии протокола для двух целей. Во-первых, они позволяют при получении первой части большого **RPC** сообщения сразу узнать его размер и выделить буфер под склейку его частей. Без них нужно было бы хранить связный список или ассоциативный массив частей, пока не собраны все.

А во-вторых, они позволяют реализовать fixed memory гарантию. Допустим, у сервера есть ограничение памяти на приём сообщений равное **M**. Клиенты посылают серверу множество **RPC** сообщений суммарного объёма **2M**. Из-за переупорядочивания **UDP** датаграмм ему прилетают вначале первые части, суммарным объёмом **M**. После этого у сервера нет никакой возможности продолжить работу, не превысив ограничение по

памяти, ведь он клиентам уже отослал **ack**-и и не может выкинуть некоторые части, чтобы допринимать вторые половины сообщений.

А поля prev_length и next_length позволяют вычислить объём каждого нового принимаемого **RPC** сообщения, и как только очередное сообщение превысит наш лимит — транспорт отказывается принимать его части и не шлёт на них подтверждения.

packet_offset позволяет вычилять весь размер окна непринятых сообщений и дополнительно лимитировать его.

TODO расписать про окно подробнее !!!!!

single_rpc_msg, multiple_rpc_msgs

В одно датаграмме может быть выставлен только один из этих флагов, либо никакой, если данных нет.

zero_padding_4_bytes, zero_padding_8_bytes

Эти поля позволяют вычислить размер padding в датаграмме. Если выставлен zero_padding_4_bytes, то к размеру надо прибавить 4 байта, если выставлено второе, то прибавить 8 байт. Они независимы, поэтому значений может быть четыре — 0, 4, 8 и 12.

payload

Если в датаграмме содержится только одна часть **RPC** сообщения, то весь payload полностью равняется её байтам.

При записи нескольких частей сообщений перед каждой дополнительно сериализуется её размер:

```
[part 0 size] [part 0 data] [part 1 size] [part 1 data] ...
```

padding

Выравнивание делается строго нулевыми байтами (и реализации должны это проверять). Его размер вычисляется через поля zero_padding_4_bytes и zero_padding_8_bytes и равен 0, 4, 8 или 12 байт.

Повторный Unencrypted заголовок

Сериализуется без magic-а и без полей flags и crypto_random. В остальном должен быть полностью равен незашифрованному **Unencrypted** заголовку.

Установка соединения

Соединение устанавливается лениво, как только у клиента появляются первые данные, которые нужно отправить. Протокол обладает свойством **0-RTT** — не имеет явного хендшейка, данные отправляются с первой же датаграммой.

Клиентская сессия имеет два состояния — ожидающая pid сервера и установленная.

Серверная сессия тоже имеет два состояния — ожидающая pid_hash от клиента и установленная.

Вначале клиент при создании попадает в первое состояние, отправляет датаграммы с полным своим local_pid, частичным remote_pid (только заполненные ip и port удалённой стороны и пустые Utime и PID) и generation (которое монотонно растёт при каждом переподключении, подробнее в разделе “TODO reconnect section name”) в **Unencrypted** заголовке, а также с version в **Encrypted** заголовке.

Как только клиент получает первую датаграмму от сервера, он запоминает local_pid оттуда и переходит в готовое состояние. С этого момента он отсылает датаграммы с pid_hash вместо local_pid, remote_pid и generation.

pid_hash высчитывается как crc64 от конкатенации байтов лексикографически меньшего pid-а с байтами большего pid-а и с байтами generation (в **Little Endian**, как и все числа в **RPC**).

Сервер, получив первую датаграмму, попадает в первое состояние, запоминает pid клиента и отправляет ему датаграммы с полными local_pid, remote_pid и присланным клиентом generation.

Как только сервер получает от клиента датаграмму с pid_hash, он переходит в готовое состояние и с этого момента отсылает датаграммы тоже только с pid_hash.

Переподключение

Если одна из сторон перезапустилась, то протокол умеет переустанавливать соединение. Для это есть специальные транспортные сообщения:

obsoletePid	0x`6f4ac134`	Сообщение с новым pid отправляющей стороны
obsoleteHash	0x7568aabd	Сообщение о том, что присланный hash больше не валиден
obsoleteGeneration	`0xb340010`	Сообщение с новым generation в соединении

	b	

Если любая сторона получает датаграмму с remote_pid, в котором utime меньше её start_time, либо utime равен её start_time и PID не равен её PID, тогда она отсылает в ответ **obsoletePid** со своим текущим pid.

Если любая сторона получает датаграмму с неизвестным ей pid_hash, то шлёт в ответ сообщение **obsoleteHash** со своим текущим pid.

Если любая сторона получает датаграмму с generation меньше известного ей в данном соединении, то отсылает в ответ сообщение **obsoleteGeneration** с известным ей generation.

Шифрование

Так же, как и в случае **TCP**, для шифрования данных используется **AES-256** в режиме **ige**. Для каждого из двух направлений общения выводится свой ключ, который может меняться, а вектор инициализации присылается в каждой датаграмме в **Unencrypted** заголовке.

При выведении ключа для шифрования выходных данных вначале составляется вектор w из конкатенации local_pid, crypto_key, remote_pid и generation, где crypto_key — общий ключ двух сторон, затем применяется формула:
$writeKey := md5.Sum(w[1:])[:12] .. sha1.Sum(w)$

Ключ для дешифрования входного потока равен ключу шифрования удалённой стороны — то есть local_pid и remote_pid меняются местами.

Поскольку клиент при установке соединения не знает полного pid сервера, он до получения первой датаграммы использует временный шифр, в котором используется частичный remote_pid — только ip и port, а Utime и PID нулевые. Соответственно, сервер до получения pid_hash от клиента использует такой же шифр.

В сообщении **ObsoleteHash** используется полный local_pid отправляющей стороны, полностью пустой remote_pid и нулевой generation.

В сообщениях **ObsoletePid** и **ObsoleteGeneration** используются local_pid и generation, известный отправляющей стороне, а remote_pid равный local_pid из сообщения, присланного удалённой стороной.

Надёжная доставка

Поскольку протокол **UDP** ненадёжный — датаграммы могут теряться, дублироваться, переупорядочиваться — транспорт сам занимается **exactly-once** доставкой надёжных **RPC** сообщений.

Для этого и существует нумерация частей через **seq-num** и отправка **ack**-ов. Как только сторона получила часть и решила её принять (она имеет право отклонить её, например, если соответствующее ей **RPC** сообщение не влезает в лимит по памяти), она обязана eventually отправить **ack** другой стороне. Поскольку датаграмма с подтверждением тоже может потеряться в сети, транспорт также обязан eventually перепосылать **ack** на часть каждый раз, когда повторно получает её.

Обычно реализации транспорта стараются совмещать отправку **ack**-ов с посылкой собственных данных. Если же данных для отправки нет, транспорт может отправить датаграмму только с подтверждениями.

В текущих реализациях на **C++** и **go** существует форсированная отправка **ack**-ов, если их накопилось 16 штук, либо по срабатыванию таймера.

В потоке частей **RPC** сообщений на отправляющей и принимающей стороне есть собственное видение доставленных и не доставленных частей:

Outgoing:
[✅] [✅] [✅] [✅] [....] [....] [✅] [....] [....] [✅] [✅] [....] [✅] [....] [....] [....] [....] [....]

Incoming:
[✅] [✅] [✅] [✅] [✅] [✅] [✅] [....] [✅] [✅] [✅] [....] [✅]

Подтверждённые сообщения на отправителе являются подмножеством полученных сообщений на принимающей стороне, так как отправитель не сразу получает **ack**-и, и они могут потеряться.

За надёжную доставку неподтверждённого суффикса отвечает отправляющая сторона. Отправитель, если имеет непустой суффикс частей без подтверждений, периодически перепосылает эти части.

За надёжную доставку пропусков отвечает принимающая сторона. Если есть пропуски, транспорт периодически перепосылает специальный resend request, в котором указывает список неполученных промежутков частей сообщений:

```
netUdpPacket.resendRange#5efaad4a
 packet_num_from:#
```

```
 packet_num_to:#
 = netUdpPacket.ResendRange;

netUdpPacket.resendRequest#643736d9
 ranges:(vector netUdpPacket.resendRange)
 = netUdpPacket.ResendRequest;
```